Theoretische Informatik 1

Vorlesungsnotizen

Prof. Dr. Roland Meyer

TU Braunschweig

Wintersemester 2024/2025

Zuletzt aktualisiert: 21. Oktober 2024

Inhaltsverzeichnis

Vorwort

Dies sind die Vorlesungsaufzeichnungen zur Vorlesung *Theoretische Informatik 1*. Seit dem 9. Februar 2017 umfasst dieses Skript den in der Vorlesung behandelten Stoff. Die handschriftlichen Notizen finden sich auf der Website unseres Instituts: `tcs.cs.tu-bs.de`. Wir geben keinerlei Garantie auf Vollständigkeit oder Korrektheit. Falls Sie einen Fehler entdecken, wenden Sie sich bitte per Email an `roland.meyer@tu-braunschweig.de`.

Pascal Reichert, Peter Chini, Mike Becker, Florian Furbach, Jürgen Koslowski, Sebastian Muskalla und Jonathan Kolberg halfen dabei, die handschriftlichen Notizen in LaTeX zu überführen. Ich bin dankbar für ihre Hilfe.

Frohes Lernen!

Roland Meyer

Literatur

Der Ausarbeitung der Vorlesung liegen die folgenden Bücher zugrunde. Die Vorlesung folgt allerdings keiner der angegebenen Quellen streng.

Teil I. Fixpunkte und Datenflussanalyse:

- F. Nielson, H. R. Nielson, C. Hankin
  Principles of Program Analysis
  Springer, 2005
- U. P. Khedker, A. Sanyal, B. Karkare
  Data Flow Analysis – Theory and Practice
  CRC Press, 2009
- H. Seidl, R. Wilhelm, S. Hack
  Übersetzerbau – Analyse und Transformation
  Springer, 2010
- R. Berghammer
  Ordnungen, Verbände und Relationen mit Anwendungen
  Springer, 2012
- G. Grätzer
  General Lattice Theory
  Birkhäuser, 2003
- G. Birkhoff
  Lattice Theory
  Providence, 1967

Teil II. Reguläre Sprachen & Teil III. Kontextfreie Sprachen:

- U. Schöning
  Theoretische Informatik – kurz gefasst
  Springer Spektrum, 2008
- J. E. Hopcroft, R. Motwani, J. D. Ullman
  Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie
  Addison-Wesley Longman, 2002
- M. Nebel
  Formale Grundlagen der Programmierung
  Springer Vieweg, 2012
- M. Sipser
  Introduction to the Theory of Computation
  International Thomson Publishing, 1996
- D. Kozen
  Automata and Computability
  Springer, 1977

Motivation

1. **Als Entwickler von Programmiersprachen:** Formuliere gültige Ausführungen in Programmen, Bibliotheken oder Betriebssystemen.

$$x = y = 0$$

```
// Thread 1                    ||   // Thread 2
if x = 42 then                 ||   if y = 36 then
    y := 36;                   ||       x := 42;
fi                             ||   fi
```

$$x = 42 \wedge y = 36$$

Das obige Programm besteht aus 2 Threads, welche den jeweiligen rechten oder linken Code parallel ausführen. Für mehrläufige Programme wie dieses gibt es im Allgemeinen mehrere mögliche Programmabläufe, die unterschiedliche Ausgaben produzieren können. Bis vor kurzem gab es in Java einen möglichen Ablauf für obigen Code, der $x = 42$ und $y = 36$ produzierte.

Frage: Wie formuliert man *gültige* Programmabläufe?

2. **Als Compiler:** Optimiere gegebenen Programmcode. Ziehe zum Beispiel konstante Zuweisungen aus Schleifen heraus:

```
// ... omitted code
while (...) do
    // ... omitted code
    x := const;
    // ... omitted code
od
```

wird optimiert zu

```
// ... omitted code
x := const; //optimiert
while (...) do
    // ... omitted code
od
```

Kritik: Wann sind Werte jemals konstant?

Antwort: Selten! Aber *Teilausdrücke* sind an manchen Programmpunkten sicher schon berechnet worden. Diese könnten wiederverwendet werden, ohne eine Neuberechnung des Ausdrucks.

Fragen:

- Wie bestimmt man diese Teilausdrücke?
- Wie argumentiert man, dass Optimierungen gültig sind?

3. **Als Schnittstelle zur Datenbank: Little Bobby Tables.** Stelle sicher, dass alle Eingaben zur Datenbank (z.B über ein Webinterface) *sanitized* sind, das heißt, sicher sind:

   Name: Robert; DROP TABLE Students;

   Wird die Eingabe des Benutzers nicht sanitized, kann die Tabelle "Students" ungewollt aus der Datenbank gelöscht werden.

   Frage: Wie implementiert man einen *Input-Sanitizer*?

Techniken, um alle drei Fragen zu beantworten, werden in der Theoretischen Informatik behandelt.

1. Um 1 und 2 zu beantworten, sind Fixpunkte und Verbände zentrale Konzepte. Die gültigen Programmabläufe lassen sich als (kleinsten) Fixpunkt formulieren, während die optimierbaren Teilausdrücke mit Hilfe einer Fixpunktberechnung bestimmt werden können.
2. Die Frage 3 ist ein sogenanntes Membership/Wortproblem in regulären Sprachen. Dieses Problem kann mit Hilfe von endlichen Automaten gelöst werden.

Teil I.

Fixpunkte und Datenflussanalyse

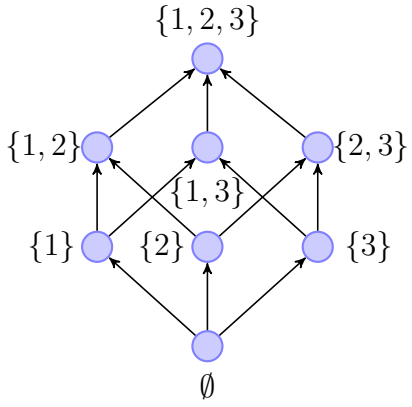
1. Verbände, Fixpunkte und die Sätze von Knaster und Tarski & Kleene

1.1. Partielle Ordnungen & Verbände

- $(\mathbb{N}, \leq)$ ist total geordnet: jeweils zwei Elemente sind in der Ordnung vergleichbar
- Einige Domänen sind nur partiell geordnet

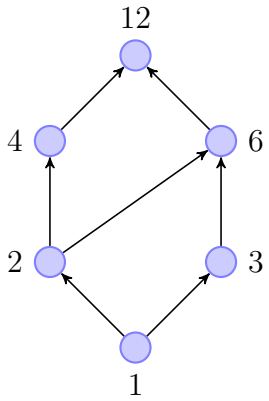
Beispiel 1.1 (Teilmengen von $\{1,2,3\}$ & Teiler von 12)**.**

Teilmengen von $\{1,2,3\}$



$\{1,2\}$ und $\{2,3\}$ sind unvergleichbar

Teiler von 12



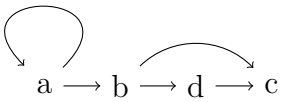
2 und 3 sind unvergleichbar.

Definition 1.2. Eine *partielle Ordnung* $(D, \leq)$ besteht aus einer nicht-leeren Menge $D \neq \emptyset$ und einer Relation $\leq \subseteq D \times D$, die die folgenden Eigenschaften hat:

- *Reflexivität*: $\forall d \in D\colon d \leq d$,
- *Transitivität*: $\forall d, d', d'' \in D\colon$ Wenn $d \leq d'$ und $d' \leq d''$, dann $d \leq d''$,
- *Antisymmetrie*: $\forall d, d' \in D\colon$ Wenn $d \leq d'$ und $d' \leq d$, dann $d = d'$.

Binäre Relationen lassen sich als *gerichtete Graphen* auffassen, z.B.

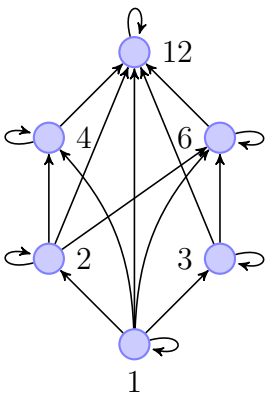
$$\{(a,a),(a,b),(b,c),(b,d),(d,c)\} =$$



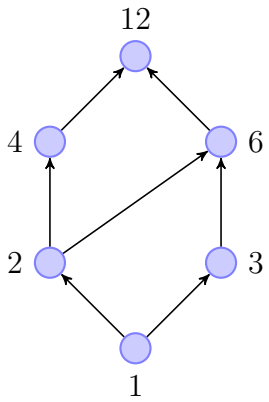
Partielle Ordnungen liefern besondere Graphen:

- Reflexivität = An jedem Knoten ist eine Schleife.
- Antisymmetrie = keine nicht-trivialen Kreise.
- Transitivität = Transitivität der Kanten.

Beispiel 1.3 (Teiler von 12)**.** Im *Hasse-Diagramm* lässt man Schleifen und durch Transitivität induzierte Kanten weg.



Gerichteter Graph



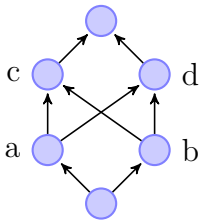
Hasse-Diagramm

Definition 1.4 (Join und Meet)**.** Sei $(D, \leq)$ eine partielle Ordnung und $X \subseteq D$ eine Teilmenge.

- Ein Element $o \in D$ heißt *obere Schranke* von X falls $x \leq o$ für alle $x \in X$.
- Ein Element $o \in D$ heißt *kleinste obere Schranke* von X (auch *Join* von X oder *Supremum* von X; Notation: $o = \bigsqcup X$), falls
  - o ist obere Schranke von X und
  - $o \leq o'$ für alle oberen Schranken o' von X.
- Ein Element $u \in D$ heißt *untere Schranke* von X falls $u \leq x$ für alle $x \in X$.
- Ein Element $u \in D$ heißt *größte untere Schranke* von X (auch *Meet* von X oder *Infimum* von X; Notation: $u = \bigsqcap X$), falls
  - u ist untere Schranke von X und
  - $u' \leq u$ für alle unteren Schranken u' von X.

Aus der Definition folgt, dass Join und Meet eindeutig sind, falls sie existieren. Angenommen sowohl o als auch o' sind kleinste obere Schranken. Dann gilt nach der zweiten definierenden Eigenschaft $o \leq o'$ und $o' \leq o$. Mit Antisymmetrie folgt $o = o'$.

Beispiel 1.5.



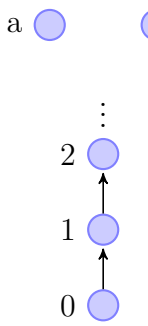
a und b haben

- c und d als obere Schranken
- aber keine kleinste obere Schranke

Definition 1.6 (Verband)**.**

- Ein *Verband* ist eine partielle Ordnung $(D, \leq)$ in der für jedes Paar $a, b \in D$ von Elementen Join $a \sqcup b$ und Meet $a \sqcap b$ existieren. Dabei ist $a \sqcup b$ Infixnotation für $\bigsqcup\{a, b\}$.
- Ein Verband heißt *vollständig*, falls für jede Teilmenge $X \subseteq D$ von Elementen Join $\bigsqcup X$ und Meet $\bigsqcap X$ existieren.

Beispiel 1.7.



kein Verband

kein vollständiger Verband

Lemma 1.8. a) Ein vollständiger Verband $(D, \leq)$ hat ein eindeutiges kleinstes Element, genannt *Bottom*:

$$\bot = \bigsqcup \emptyset = \bigsqcap D \ .$$

b) Ein vollständiger Verband hat ein eindeutiges größtes Element, genannt *Top*:

$$\top = \bigsqcap \emptyset = \bigsqcup D \ .$$

c) Jeder endliche Verband $(D, \leq)$, d.h. ein Verband mit endlicher Menge D, ist bereits vollständig

1.2. Monotone Funktionen & der Satz von Knaster und Tarski

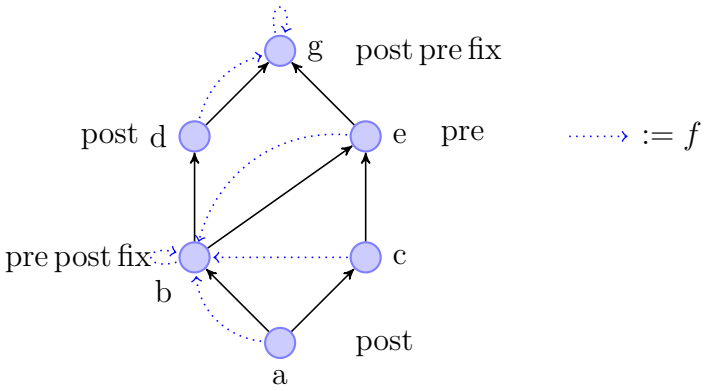
Definition 1.9 (Monotone Funktionen und Fixpunkte)**.** Sei $(D, \leq)$ eine partielle Ordnung und $f : D \to D$ eine Funktion.

- Die Funktion f heißt *monoton*, falls

$$\forall x, y \in D\colon \text{Wenn } x \leq y, \text{ dann } f(x) \leq f(y) \ .$$

- Ein *Fixpunkt* von f ist ein Element $x \in D$ mit $f(x) = x$.
- Ein *Pre-Fixpunkt* von f ist ein Element $x \in D$ mit $f(x) \leq x$.
- Ein *Post-Fixpunkt* von f ist ein Element $x \in D$ mit $x \leq f(x)$.

Beispiel 1.10.



Beachte: Fixpunkte sind nahe verwandt mit Nullstellen (wenn es denn Nullen, Addition etc. gibt).

Von Nullstellen zu Fixpunkten:

$$f(x) = 0 \iff f(x) + x = x \iff (f + \mathrm{id})(x) = x$$

Die Nullstellen von f sind genau die Fixpunkte von $(f + \mathrm{id})$.

Von Fixpunkten zu Nullstellen:

$$f(x) = x \iff f(x) - x = 0 \iff (f - \mathrm{id})(x) = 0$$

Die Fixpunkte von f sind genau die Nullstellen von $(f - \mathrm{id})$.

Satz 1.11 (Knaster und Tarski, 1955)**.** Sei $(D, \leq)$ ein vollständiger Verband und $f : D \to D$ monoton.

a) Dann besitzt f einen (eindeutigen) *kleinsten* Fixpunkt, gegeben durch

$$\mathrm{lfp}(f) = \bigsqcap \mathrm{Prefix}(f) \ .$$

b) Ferner besitzt f einen (eindeutigen) *größten* Fixpunkt, gegeben durch

$$\text{gfp}(f) = \bigsqcup \text{Postfix}(f) \ .$$

Beweis. Zeige die Behauptung für $\text{lfp}(f)$, der Beweis für $\text{gfp}(f)$ ist analog.

Sei $\ell = \bigsqcap \text{Prefix}(f)$.

Zeige zunächst $f(\ell) \leq \ell$.

Da $\ell \leq \ell'$ für alle $\ell' \in \text{Prefix}(f)$, und da f monoton, folgt

$$f(\ell) \leq f(\ell') \leq \ell' \text{ für alle } \ell' \in \text{Prefix}(f) \ .$$

Da $\ell = \bigsqcap \text{Prefix}(f)$ folgt

$$f(\ell) \leq \ell \ . \tag{*}$$

Zeige nun $\ell \leq f(\ell)$.

Mit (*) gilt: $f(f(\ell)) \leq f(\ell)$.

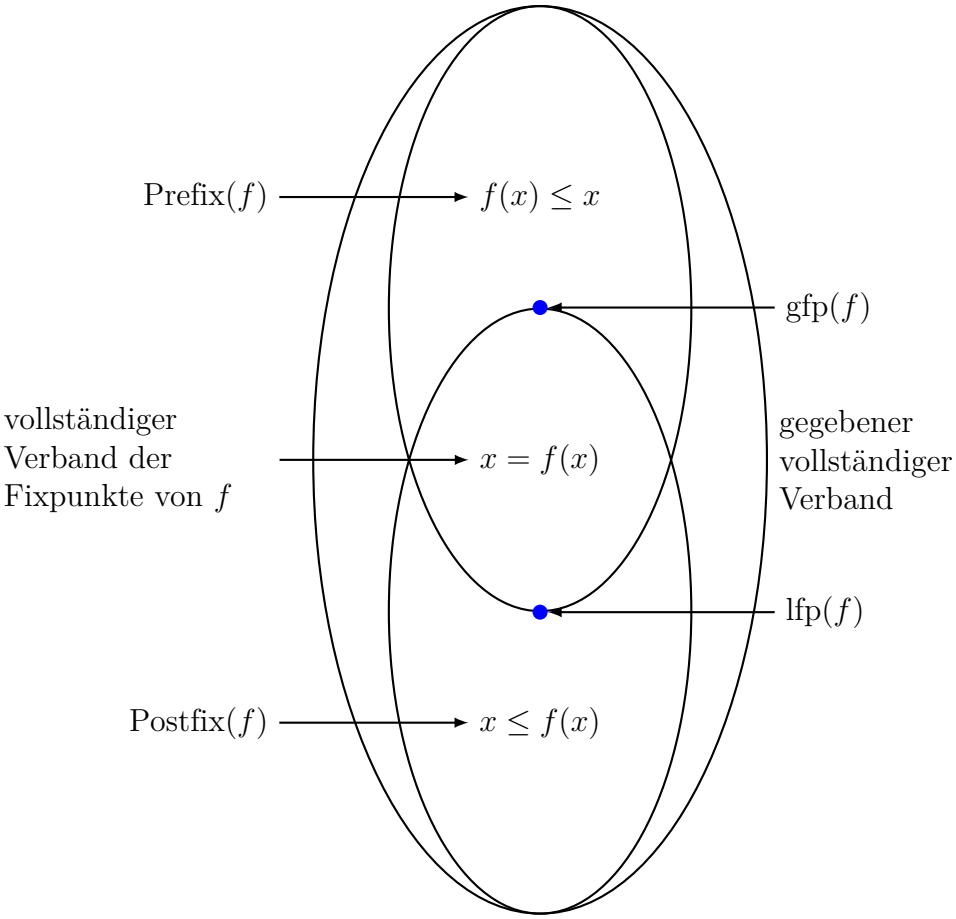
Damit gilt

$$f(\ell) \in \text{Prefix}(f) \text{ und so } \ell \leq f(\ell) \ . \tag{**}$$

Mit Antisymmetrie folgt aus (*) und (**)

$$\ell = f(\ell)$$

Damit ist gezeigt, dass ℓ ein Fixpunkt ist. Beachte, dass jeder Fixpunkt von f auch ein Prefixpunkt ist und daher in $\text{Prefix}(f)$ enthalten ist. Da ℓ als kleinste untere Schranke aller Prefixpunkte definiert war, ist ℓ insbesondere kleiner als jeder andere Fixpunkt und damit der kleinste Fixpunkt. □



1.3. Ketten & der Satz von Kleene

Definition 1.12 (Kette)**.** Sei $(D, \leq)$ eine partielle Ordnung.

- Eine Teilmenge $K \subseteq D$ heißt *Kette* wenn sie total geordnet ist:

$$\forall k_1, k_2 \in K : k_1 \leq k_2 \text{ oder } k_2 \leq k_1 \ .$$

- Eine Folge $(k_i)_{i \in \mathbb{N}}$ heißt *aufsteigende Kette*, falls

$$k_i \leq k_{i+1} \text{ für alle } i \in \mathbb{N} \ .$$

- Eine Folge $(k_i)_{i \in \mathbb{N}}$ heißt *absteigende Kette*, falls

$$k_i \geq k_{i+1} \text{ für alle } i \in \mathbb{N} \ .$$

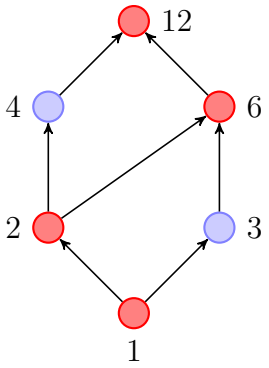
- Eine auf-/absteigende Kette $(k_i)_{i \in \mathbb{N}}$ wird *stationär*, falls

$$\exists n \in \mathbb{N} \colon \forall i \geq n \colon k_i = k_n \ .$$

- $(D, \leq)$ hat *endliche Höhe*, falls jede Kette K in D endlich viele Elemente hat.
- $(D, \leq)$ hat *beschränkte Höhe*, falls es eine Schranke $n \in \mathbb{N}$ gibt, so dass jede Kette höchstens n Elemente hat.

Beispiel 1.13.

- In $(\mathbb{N}, \leq)$ wird jede absteigende Kette stationär. Es gibt jedoch unendliche echt aufsteigende Ketten.
- Die rot markierten Knoten bilden eine Kette.



Definition 1.14 (Kettenbedingung)**.** Eine partielle Ordnung $(D, \leq)$

- erfüllt die *aufsteigende Kettenbedingung (ACC)*[1], falls jede aufsteigende Kette $k_0 \leq k_1 \leq \ldots$ stationär wird. (Man sagt auch $(D, \leq)$ ist *artinsch*, nach Emil Artin.)
- erfüllt die *absteigende Kettenbedingung (DCC)*[2] falls jede absteigende Kette $k_0 \geq k_1 \geq \ldots$ stationär wird. (Man sagt auch $(D, \leq)$ ist *noethersch*, nach Emmy Noether.)

Lemma 1.15. Eine partielle Ordnung hat endliche Höhe gdw. (ACC) und (DCC) erfüllt sind

Definition 1.16 (Stetigkeit)**.** Sei $(D, \leq)$ ein vollständiger Verband. Eine Funktion $f : D \to D$ heißt

- $\sqcup$-*stetig* (aufwärtsstetig), falls für jede Kette K in D gilt

$$\begin{aligned} f\left(\bigsqcup K\right) &= \bigsqcup f(K) \\ &= \bigsqcup \{f(k) \mid k \in K\} \ . \end{aligned}$$

- $\sqcap$-*stetig* (abwärtsstetig), falls für jede Kette K in D gilt

$$\begin{aligned} f\left(\bigsqcap K\right) &= \bigsqcap f(K) \\ &= \bigsqcap \{f(k) \mid k \in K\} \ . \end{aligned}$$

Satz 1.17 (Monotonie impliziert Stetigkeit)**.** Sei $(D, \leq)$ ein vollständiger Verband und $f : D \to D$ monoton.

[1] ascending chain condition
[2] descending chain condition

a) Falls $(D, \leq)$ (ACC) erfüllt, dann ist f $\sqcup$-stetig.

b) Falls $(D, \leq)$ (DCC) erfüllt, dann ist f $\sqcap$-stetig.

Beweis. Wir zeigen a). Der Beweis von b) geht analog.

Sei K eine Kette in D. Es ist zu zeigen: $f\left(\bigsqcup K\right) = \bigsqcup f(K)$.

Wir beweisen $f\left(\bigsqcup K\right) \leq \bigsqcup f(K)$ und $f\left(\bigsqcup K\right) \geq \bigsqcup f(K)$, dann folgt die gewünschte Aussage mit Antisymmetrie.

" $\leq$ " Für alle $k \in K$: $k \leq \bigsqcup K$.
Wegen Monotonie damit auch $f(k) \leq f\left(\bigsqcup K\right)$.
Da dies für alle k gilt, gilt auch $\bigsqcup f(K) \leq f\left(\bigsqcup K\right)$.

" $\geq$ " Wir zeigen zunächst, dass es in K ein größtes Element gibt, d.h. es existiert $k' \in K$, so dass für alle $k \in K$ gilt: $k \leq k'$.

Angenommen dies ist nicht der Fall, d.h. für alle k' gibt es ein $k'' \in K$, so dass k' und k'' unvergleichbar sind oder $k'' > k'$ gilt. Da alle Elemente einer Kette vergleichbar sind, kann der erste Fall nie eintreten. Unter der Annahme, dass es zu jedem Element ein echt größeres gibt, können wir aber eine unendliche echt aufsteigende Kette konstruieren. Dies ist ein Widerspruch zur aufsteigenden Kettenbedingung (ACC).

Es gibt also ein größtes Element k' in der Kette. Damit gilt

$$f\left(\bigsqcup K\right) = f(k') \leq \bigsqcup f(K).$$

□

Lemma 1.18. Sei $(D, \leq)$ ein vollständiger Verband und $f : D \to D$ monoton.

Die Folge

$$\left(f^i(\bot)\right)_{i \in \mathbb{N}}$$

mit

$$\begin{aligned} f^0(\bot) &= \bot \ , \\ f^{i+1}(\bot) &= f\left(f^i(\bot)\right) \ , \end{aligned}$$

ist eine aufsteigende Kette.

Beweis. Wir zeigen $f^i(\bot) \leq f^{i+1}(\bot)$ für alle $i \in \mathbb{N}$ per Induktion.

IA: $f^0(\bot) = \bot \leq f(\bot)$, da $\bot = \bigsqcap D$.

IV: Gelte $f^i(\bot) \leq f^{i+1}(\bot)$ für ein i.

IS:
$$\begin{array}{lll} f^{i+1}(\bot) & = & f(f^i(\bot)) \\ & \underset{\text{IV + Monotonie}}{\leq} & f(f^{i+1}(\bot)) = f^{i+2}(\bot)\ . \end{array}$$

□

Satz 1.19 (Kleene)**.** Sei $(D, \leq)$ ein vollständiger Verband und $f : D \to D$ monoton.

a) Ist f $\sqcup$-stetig, dann gilt

$$\text{lfp}(f) = \bigsqcup \{ f^i(\bot) \mid i \in \mathbb{N} \}$$

b) Ist f $\sqcap$-stetig, dann gilt

$$\text{gfp}(f) = \bigsqcap \{ f^i(\top) \mid i \in \mathbb{N} \}$$

Beweis. Wir zeigen a). Der Beweis von b) funktioniert analog.

Zeige: $\bigsqcup \{ f^i(\bot) \mid i \in \mathbb{N} \}$ ist Fixpunkt.

$$\begin{array}{rcl} & & f\left(\bigsqcup \{ f^i(\bot) \mid i \in \mathbb{N} \}\right) \\ (f \sqcup\text{-stetig}) & = & \bigsqcup \{ f^{i+1}(\bot) \mid i \in \mathbb{N} \} \\ (\bot = \bigsqcap D) & = & \bigsqcup \{ f^i(\bot) \mid i \in \mathbb{N} \} \end{array}$$

Zeige: $\bigsqcup \{ f^i(\bot) \mid i \in \mathbb{N} \}$ ist kleinster Fixpunkt.

- Betrachte $d \in D$ mit $f(d) = d$ und zeige $\bigsqcup \{ f^i(\bot) \mid i \in \mathbb{N} \}$ ist kleiner
- Induktion nach $i \in \mathbb{N}$ gibt $f^i(\bot) \leq d$ für alle $i \in \mathbb{N}$.

  IA: $f^0(\bot) = \bot \leq d$, da $\bot = \sqcap D$.

  IV: Angenommen $f^i(\bot) \leq d$ für ein i.

  IV:
  $$f^{i+1}(\bot) = f\left(f^i(\bot)\right) \overset{\text{IV+Mon.}}{\leq} f(d) \overset{\text{Vor.}}{=} d\ .$$

- Da $f^i(\bot) \leq d$ für alle $i \in \mathbb{N}$ folgt
  $$\bigsqcup \{ f^i(\bot) \mid i \in \mathbb{N} \} \leq d$$

□

Satz 1.20. Sei $(D, \leq)$ ein vollständiger Verband mit (ACC) und (DCC).

Sei $f : D \to D$ monoton.

Dann ist

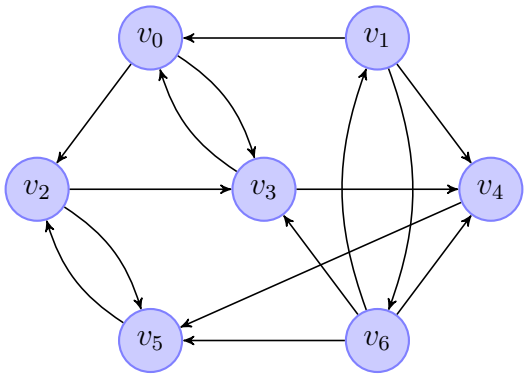
$$\begin{aligned} \mathrm{lfp}(f) &= \bigsqcup \{f^i(\bot) \mid i \in \mathbb{N}\} \\ &= f^n(\bot) \qquad \text{für ein } n \in \mathbb{N} \text{ mit } f^n(\bot) = f^{n+1}(\bot), \end{aligned}$$

$$\begin{aligned} \mathrm{gfp}(f) &= \bigsqcap \{f^i(\top) \mid i \in \mathbb{N}\} \\ &= f^n(\top) \qquad \text{für ein } n \in \mathbb{N} \text{ mit } f^n(\top) = f^{n+1}(\top). \end{aligned}$$

Beweis. Aus Monotonie folgt Stetigkeit wegen (ACC) und (DCC), die Aussage folgt dann mit dem Satz von Kleene. □

Beispiel 1.21. Betrachte den folgenden Graphen $G = (V, E)$ und die monotone Funktion $f : \mathcal{P}(V) \to \mathcal{P}(V)$ auf dem vollständigen Verband $(\mathcal{P}(V), \subseteq)$ mit

$$f(X) := \{v_0\} \cup \{v \in V \mid \exists x \in X \colon (x, v) \in E\}$$



Zeige: (ACC) (bzw. (DCC)) ist erfüllt. Der Verband ist mit 128 Elementen endlich und hat daher beschränkte Höhe. Sowohl aufsteigende, als auch absteigende Ketten werden diese Schranke einhalten und höchstens 127-mal echten Fortschritt verzeichnen, bevor sie stabil werden. Damit sind sowohl (ACC) als auch (DCC) erfüllt.

Zeige: f ist monoton. Seien $X \in \mathcal{P}(V)$ und $Y \in \mathcal{P}(V)$ zwei Elemente des Verbands, sodass $X \subseteq Y$ gilt. Zu zeigen ist $f(X) \subseteq f(Y)$. Sei $z \in f(X)$. Es gilt $z = v_0$ oder die Existenz eines $x \in X$ mit $(x, v) \in E$. Im ersten Fall folgt $z \in f(Y)$ direkt aus der Definition. Im anderen Fall ist $x \in Y$ auch ein passender Kandidat für die rechte Seite in $f(Y)$. Damit gilt $z \in f(Y)$ für alle $z \in f(X)$.

$f^0(\bot) = \bot = \emptyset$

$f^1(\bot) = \{v_0\} \cup \{v \in V \mid \exists x \in f^0(\bot) \colon (x, v) \in E\} = \{q_0\} \cup \emptyset = \{q_0\}$

$f^2(\bot) = \{v_0\} \cup \{v \in V \mid \exists x \in f^1(\bot) \colon (x, v) \in E\} = \{q_0\} \cup \{v_2, v_3\} = \{v_0, v_2, v_3\}$

$f^3(\bot) = \{v_0\} \cup \{v \in V \mid \exists x \in f^2(\bot) \colon (x, v) \in E\} = \{v_0, v_2, v_3, v_4, v_5\}$

$f^4(\bot) = \{v_0\} \cup \{v \in V \mid \exists x \in f^3(\bot) \colon (x, v) \in E\} = \{v_0, v_2, v_3, v_4, v_5\}$

Nachdem die Folge sich stabilisiert hat, können wir $\mathrm{lfp}(f) = f^3(\bot)$ als die Menge aller von v_0 erreichbaren Knoten in G identifizieren.

2. Datenflussanalyse

Ziel: Analysiere das Verhalten von Programmen *statisch*, d.h. zur Compile-Zeit

Ansatz: Fixpunktberechnung auf einer abstrakten Domäne

2.1. While-Programme

Definition 2.1 (Syntax beschrifteter While-Programme)**.** Die *Syntax von beschrifteten While-Programmen* ist durch folgende EBNF gegeben:

$$
\begin{aligned}
a \quad &::= \quad k \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \cdot a_2 \\
&\qquad // \text{ Arithmetische Ausdrücke, repräsentieren ganze Zahlen} \\
b \quad &::= \quad t \mid a_1 = a_2 \mid a_1 > a_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \\
&\qquad // \text{ Bool'sche Ausdrücke} \\
c \quad &::= \quad [\text{skip}]^\ell \mid [x := a]^\ell \mid c_1; c_2 \\
&\qquad \mid \text{ if } [b]^\ell \text{ then } c_1 \text{ else } c_2 \text{ endif} \\
&\qquad \mid \text{ while } [b]^\ell \text{ do } c \text{ endwhile} \\
&\qquad // \text{ Programme, jeder Befehl hat ein Label } \ell
\end{aligned}
$$

- Hierbei ist $k \in \mathbb{Z}, t \in \mathbb{B} = \{0,1\} = \{\mathsf{false}, \mathsf{true}\}$ und $x \in \text{Var}$ eine Variable.
- Wir nehmen an, dass alle Labels im Programm verschieden sind.
- Beschriftete Befehle werden *Blöcke* genannt.

Programme lassen sich als *Kontrollflussgraphen* $G = (B, E, F)$ darstellen, das heißt als gerichtete Graphen mit Knotenmenge

$$B = \textit{Blöcke} \text{ im Programm },$$

in der – je nach Analyse – die initialen oder finalen Blöcke

$$E = \text{Menge an } \textit{extremalen Blöcken} \subseteq B$$

besonders betrachtet werden. Die Kanten des Graphen sind durch die sogenannte *Flussrelation*

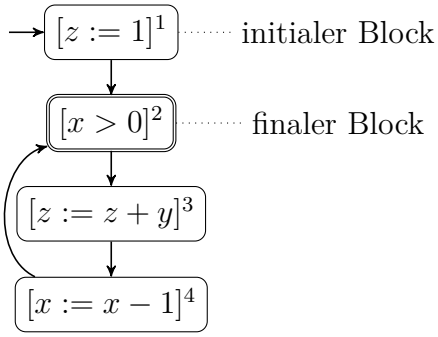
$$F \subseteq B \times B$$

gegeben.

- Typischerweise repräsentieren Kontrollflussgraphen die Struktur eines Programms

  Beispielsweise korrespondiert das links stehende Programm zum rechts angegebenen Kontrollflussgraphen.

```
c = [z := 1]^1;
while [x > 0]^2 do
    [z := z+y]^3;
    [x := x-1]^4
endwhile
```



- Es gibt jedoch Datenflussanalysen, die Programme entgegen der Befehlsfolge (rückwärts) analysieren (Live-Variables zum Beispiel). Daher werden wir bei einer Datenflussanalyse den zugrundeliegenden Kontrollflussgraphen genau festlegen.
- Für Kontrollflussgraphen wird – je nach Analyse – angenommen,
  - dass der initiale Block keine eingehenden Kanten hat, bzw.
  - dass die finalen Blöcke keine ausgehenden Kanten haben.

  Diese Form lässt sich durch Hinzufügen von `skip`-Befehlen immer herstellen. Das obige Beispiel erfüllt die Bedingung für initiale Blöcke, verletzt aber die Bedingung für finale Blöcke.

2.2. Monotone Frameworks

Monotone Frameworks nutzen einen vollständigen Verband als abstrakte Datendomäne und imitieren die Befehle des Programms durch monotone Funktionen.

Definition 2.2 (Datenflusssystem)**.** Ein *Datenflusssystem* ist ein Tupel

$$S = (G, (D, \leq), i, TF)$$

- $G = (B, E, F)$ ein *Kontrollflussgraph*,
- $(D, \leq)$ ein *vollständiger Verband* mit (ACC),
- $i \in D$ ein *Anfangswert* für Extremalblöcke,
- $TF = \{f_b : D \to D \mid b \in B\}$ eine Familie von *Transferfunktionen*, eine für jeden Block, die alle *monoton* sind.

Bemerkung: Falls man einen vollständigen Verband $(D, \leq)$ benutzen möchte, in dem (DCC) gilt, kann man den dualen Verband $(D, \geq)$ verwenden, in dem dann (ACC) gilt.

Ein Datenflusssystem induziert ein *Gleichungssystem* mit einer Variable X_b pro Block $b \in B$ und jeweils der definierenden Gleichung

$$X_b = \begin{cases} i & \text{, falls } b \in E, \\ \bigsqcup \{ f_{b'} (X_{b'}) \mid (b', b) \in F \} & \text{, sonst.} \end{cases}$$

Das bedeutet, dass Extremalblöcke durch den spezifizierten Initialwert repräsentiert werden und alle anderen Blöcke durch den Join der Werte, die man durch die eingehenden Kanten erhält. Intuitiv repräsentiert X_b also den Datenflusswert am Eingang des Blocks b.

Ein Vektor $(d_1, \ldots, d_{|B|}) \in D^{|B|}$ heißt *Lösung von S*, falls

$$d_b = \begin{cases} i & \text{, falls } b \in E, \\ \bigsqcup \{ f_{b'} (d_{b'}) \mid (b', b) \in F \} & \text{, sonst.} \end{cases}$$

Um den Zusammenhang zwischen den Lösungen des Gleichungssystems von S sowie Fixpunkten herzustellen, definieren wir die Funktion

$$\begin{array}{rccc} g_s & : & D^{|B|} & \to & D^{|B|} \\ & & (d_1, \ldots, d_{|B|}) & \mapsto & (d'_1, \ldots, d'_{|B|}) \end{array}$$

mit

$$d'_b = \begin{cases} i & \text{, falls } b \in E, \\ \bigsqcup \{ f_{b'} (d_{b'}) \mid (b', b) \in F \} & \text{, sonst.} \end{cases}$$

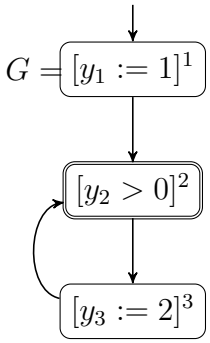
Satz 2.3. Vektor $\bar{d} = (d_1, \ldots, d_{|B|}) \in D^{|B|}$ ist (kleinste) Lösung des von S induzierten Gleichungssystem gdw. $\bar{d}$ ist (kleinster) Fixpunkt von g_s, d.h. $g_s(\bar{d}) = \bar{d}$.

Bemerkung: Der kleinste Fixpunkte kann gemäß dem Satz von Kleene mittels Iteration berechnet werden.

Bemerkung: Wir interessieren uns für die kleinste Lösung, da diese die präziseste Information liefert.

Beispiel 2.4. Es soll eine Programmanalyse definiert werden, die die Menge an Variablen berechnet, die an einem Programmpunkt geschrieben worden sind. Betrachte das linksstehende Programm, der zugehörige Kontrollflussgraph ist rechts angegeben.

```
c = [y1:= 1]1;
while [y2 >0]2 do
    [y3:= 2]3;
endwhile
```



Das zugehörige Datenflusssystem ist

$$S = (G, (\mathcal{P}(\{y_1, y_2, y_3\}, \subseteq)), \emptyset, \{f_1, f_2, f_3\})$$

mit

$$\begin{aligned} f_1, f_2, f_3 : \mathcal{P}(\{y_1, y_2, y_3\}) &\to \mathcal{P}(\{y_1, y_2, y_3\}) \\ f_1(X) &= X \cup \{y_1\} \\ f_2(X) &= X \\ f_3(X) &= X \cup \{y_3\} \end{aligned}$$

Das Datenflusssystem induziert das Gleichungssystem

$$\begin{aligned} X_1 &= \emptyset \\ X_2 &= \underbrace{X_1 \cup \{y_1\}}_{=f_1(X_1)} \cup \underbrace{X_3 \cup \{y_3\}}_{f_3(X_3)} \\ X_3 &= \underbrace{X_2}_{f_2(X_2)} \end{aligned}$$

Eine Lösung ist $(\emptyset, \{y_1, y_3\}, \{y_1, y_3\})$.

2.3. Beispiele für Datenflussanalysen

Klassifikation von Datenflussanalysen Datenflussanalysen lassen sich anhand der vier folgenden Parameter klassifizieren.

Richtung der Analyse:

Vorwärts: Berechne Information über die Vergangenheit von Daten.

> Das heißt die initialen Blöcke werden als extremal angesehen und die Analyse wird in Programmrichtung durchgeführt.

Rückwärts: Berechne Information über das zukünftige Verhalten von Daten.

> Hierzu werden die finalen Blöcke als extrem angesehen, und die Analyse wird gegen Programmrichtung durchgeführt. Hierzu dreht man im Kontrollflussgraphen alle Kanten um.

Approximation der Information

May: Überapproximiere die Information über Daten.

> May-Analysen spiegeln jede Information wider, die (möglicherweise) in einem realen Ablauf eintreten kann. Damit können May-Informationen nicht verletzt werden. Allerdings ist nicht garantiert, dass eine Information auch in einem realen Ablauf erreicht wird.

Must: Unterapproximiere die Information über Daten.

> Must-Analysen spiegeln nur Information wider, die definitiv in jedem realen Ablauf eintritt. Damit liefern Must-Analysen verlässlich eintretende Informationen. Allerdings geben Must-Analysen nicht alle eventuell eintretenden Informationen wieder.

Berücksichtigung von Prozeduren

Intraprozedural: Analyse einer einzelnen Prozedur, typischerweise `main`.

> Um Programme intraprozedural zu analysieren, nutze *Inlining*, d.h. das Ersetzen von Prozeduraufrufen durch den Rumpf. Inlining ist bei Rekursion nicht möglich, Intraprozedurale Analysen unterstützen keine Rekursion.

Interprozedural: Analyse eines ganzen Programms mit Rekursion.

Berücksichtigung des Kontrollflusses:

Control-flow sensitive: Berücksichtige die Anordnung der Befehle im Programm.

Die Analyse berechnet separate Information für jeden Block.

Vorteil: präzise. Nachteil: ineffizient.

Control-flow insensitive: Vergiss die Anordnung der Befehle im Programm.

Die Analyse berechnet eine Information für alle Blöcke.

Vorteil: effizient. Nachteil: unpräzise.

Wir betrachten vier klassische Analysen, die alle vier Kombinationen aus Richtung und Approximation abdecken. Allerdings sind alle vier Analysen conrol-flow sensitiv und intraprozedural. Folgende Tabelle zeigt die Analysen und den Zusammenhang zwischen:

Richtung	$\leftrightarrow$	Wahl des Kontrollflussgraphen mit Extremalknoten
Approximation	$\leftrightarrow$	Wahl des Verbandes mit Join und Bottom.

Instanz	**Reaching-Definitions**	**Available-Expr.**	**Live-Var.**	**Busy-Expr.**
Richtung	vorwärts		rückwärts	
Extremal (E)	initaler Block		finale Blöcke	
Fluss. (F)	in Programmordnung		gegen Programmordnung	
Approx.	may	must	may	must
Verband	$(\mathcal{P}(Vars \times (Blocks \cup \{?\})), \subseteq)$	$(\mathcal{P}(AExp), \supseteq)$	$(\mathcal{P}(Vars), \subseteq)$	$(\mathcal{P}(AExp), \supseteq)$
Join ($\sqcup$)	$\cup$	$\cap$	$\cup$	$\cap$
Bottom ($\bot$)	$\emptyset$	$AExp$	$\emptyset$	$AExp$
Anfangsw. i	$\{(x, ?) \mid x \in Vars\}$	$\emptyset$	$Vars$	$\emptyset$
Transferf. TF	$f_b(X) := (X \setminus kill(b)) \cup gen(b)$			

In allen vier Analysen sind die Transferfunktion für jeden Block vom Typ

$$f_b(X) := (X \setminus kill(b)) \cup gen(b) \ .$$

Dies bedeutet, dass jeder Block b einen Teil der Werte aus dem Datenflusswert entfernt, nämlich die Werte, die nach dem Block nicht mehr gültig sind. Dies wird dadurch realisiert, dass $kill(b)$ aus der Menge X herausgenommen wird.

Zudem generiert jeder Block b neue Informationen. Dies wird durch das Hinzufügen von $gen(b)$ realisiert.

Man beachte, dass $kill(b)$ und $gen(b)$ nur vom Block b, nicht allerdings vom Datenflusswert X abhängig sind. Daher sind die Transferfunktionen dieser Form immer monoton.

2.4. Beispiel 1: Reaching-Definitions-Analyse

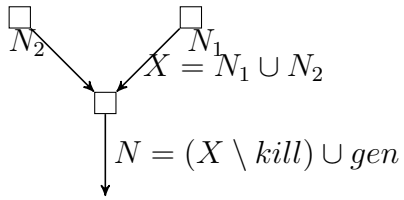
Ziel: Berechne für jeden Block die Zuweisungen, die es gegeben haben könnte.

Wir wollen für jeden Block und jede Variable berechnen, woher die letzte Zuweisungen an die entsprechende Variable in einer Ausführung, die den Block erreicht, kommen könnte.

Klassifikation:

Vorwärtsanalyse, die Information über die Vergangenheit von Daten berechnet.
May-Analyse, die das Verhalten aller einzelnen Ausführungen überapproximiert. Das heißt, das Verhalten jeder Ausführung ist sicher in der Information enthalten.

Idee:



Anwendungen: Berechnung von *Use-Definition-Chains*, die angeben, welche Zuweisungen (Definitions) von einem Block genutzt werden.

Use-Definition-Chains sind die Grundlage für *Code-Motion-Optimierungen*.

Beispiel 2.5. Betrachte ein Programm mit Variablen *Vars* und Blöcken *Blocks*.

Definiere das Datenflusssystem

$$S = (G, (D, \preceq), i, \{f_b : D \to D \mid b \in \mathit{Blocks}\}) \ .$$

- Kontrollflussgraph $G = (B, E, F)$:
  $B = \mathit{Blocks}$, $E =$ initialer Block, $F =$ Kontrollfluss in Programmordnung.
- Verband $(D, \preceq)$:
  $(D, \preceq) = (\mathcal{P}(\mathit{Vars} \times (\mathit{Blocks} \cup \{?\})), \subseteq)$.

  Es handelt sich um einen Potenzmengenverband. (ACC) gilt, da der Verband endlich ist.

  Die Bedeutung der Elemente in $\mathit{Vars} \times (\mathit{Blocks} \cup \{?\})$ ist wie folgt:

  $(x, ?)$ = x ist möglicherweise noch nicht initialisiert.
  (x, b) = x hat möglicherweise die letzte Zuweisung von Block b erhalten.

- Anfangswert i:
  $\{(x, ?) \mid x \in \mathit{Vars}\}$.
- Transferfunktionen $f_b : D \to D$:

$$f_b : \mathcal{P}(\mathit{Vars} \times (\mathit{Blocks} \cup \{?\})) \to \mathcal{P}(\mathit{Vars} \times (\mathit{Blocks} \cup \{?\}))$$
$$X \mapsto (X \setminus \mathit{kill}(b)) \cup \mathit{gen}(b)$$

  Die Mengen $\mathit{kill}(b), \mathit{gen}(b) \subseteq \mathit{Vars} \times (\mathit{Blocks} \cup \{?\})$ sind

$$\mathit{kill}(b) = \begin{cases} \{(x, ?)\} \cup \{(x, b') \mid b' \in \mathit{Blocks}\}, & \text{falls } b = [x := a]^b \\ \emptyset, & \text{sonst.} \end{cases}$$

  //Zuweisungen, die von Block b überschrieben werden.

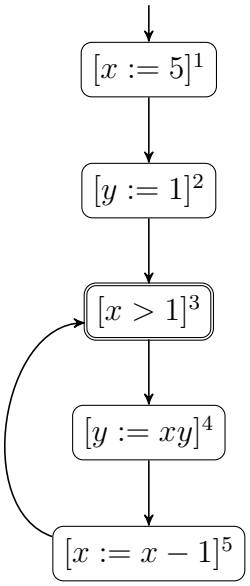
$$\mathit{gen}(b) = \begin{cases} \{(x, b)\}, & \text{falls } b = [x := a]^b \\ \emptyset, & \text{sonst.} \end{cases}$$

  //Zuweisungen, die von Block b generiert werden.

  Die Transferfunktionen sind wie oben erwähnt monoton.

Betrachte das Beispielprogramm bzw. den zugehörigen Kontrollflussgraphen.

```
[x:=5]1;
[y:=1]2;
while [x > 1]3 do
    [y:=xy]4;
    [x:=x-1]5;
endwhile
```



Die Transferfunktionen sind wie folgt.

Block	$\mathit{kill}(b)$	$\mathit{gen}(b)$	$f_b(X)$
$[x := 5]^1$	$\{(x, ?), (x, 1), (x, 5)\}$	$\{(x, 1)\}$	$(X \setminus \{(x, ?), (x, 1), (x, 5)\}) \cup \{(x, 1)\}$
$[y := 1]^2$	$\{(y, ?), (y, 2), (y, 4)\}$	$\{(y, 2)\}$	$(X \setminus \{(y, ?), (y, 2), (y, 4)\}) \cup \{(y, 2)\}$
$[x > 1]^3$	$\emptyset$	$\emptyset$	X
$[y := xy]^4$	$\{(y, ?), (y, 2), (y, 4)\}$	$\{(y, 4)\}$	$(X \setminus \{(y, ?), (y, 2), (y, 4)\}) \cup \{(y, 4)\}$
$[x := x - 1]^5$	$\{(x, ?), (x, 1), (x, 5)\}$	$\{(x, 5)\}$	$(X \setminus \{(x, ?), (x, 1), (x, 5)\}) \cup \{(x, 5)\}$

In der Tabelle sind die $kill(b)$ Mengen auf die Blöcke eingeschränkt worden, die eine Zuweisung auf die jeweilige Variable durchführen.

Das folgende Gleichungssystem wird vom Datenflusssystem induziert.

$$X_1 = \underbrace{\{(x,?),(y,?)\}}_{=i}$$
$$X_2 = \underbrace{(X_1 \setminus \{(x,?),(x,1),(x,5)\}) \cup \{(x,1)\}}_{=f_1(X_1)}$$
$$X_3 = \underbrace{((X_2 \setminus \{(y,?),(y,2),(y,4)\}) \cup \{(y,2)\})}_{=f_2(X_2)} \cup \underbrace{((X_5 \setminus \{(x,?),(x,1),(x,5)\}) \cup \{(x,5)\})}_{=f_5(X_5)}$$
$$X_4 = X_3$$
$$X_5 = (X_4 \setminus \{(y,?),(y,2),(y,4)\}) \cup \{(y,4)\}$$

Wir berechnen eine Lösung des Gleichungssystems durch Iteration von

$$g_S : \mathcal{P}\left(Vars \times (Blocks \cup \{?\})\right)^5 \to \mathcal{P}\left(Vars \times (Blocks \cup \{?\})\right)^5$$

auf $\bot$ von $(\mathcal{P}\left(Vars \times (Blocks \cup \{?\})\right)^5, \subseteq^5)$ bis zum kleinsten Fixpunkt:

Iter.	$(d_1$	d_2	d_3	d_4	$d_5)$
$g_S^0(\bot)$	$(\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset)$
$g_S^1(\bot)$	$(\{(x,?),(y,?)\}$	$\{(x,1)\}$	$\{(y,2),(x,5)\}$	$\emptyset$	$\{(y,4)\})$
$g_S^2(\bot)$	$(\{(x,?),(y,?)\}$	$\{(y,?),(x,1)\}$	$\{(x,1),(y,2),(y,4),(x,5)\}$	$\{(y,2),(x,5)\}$	$\{(y,4)\})$
$g_S^3(\bot)$	$(\{(x,?),(y,?)\}$	$\{(y,?),(x,1)\}$	$\{(x,1),(y,2),(y,4),(x,5)\}$	$\{(x,1),(y,2),(y,4),(x,5)\}$	$\{(x,5)(y,4)\})$
$g_S^4(\bot)$	$(\{(x,?),(y,?)\}$	$\{(y,?),(x,1)\}$	$\{(x,1),(y,2),(y,4),(x,5)\}$	$\{(x,1),(y,2),(y,4),(x,5)\}$	$\{(x,1),(x,5),(y,4)\})$
$g_S^5(\bot)$	$(\{(x,?),(y,?)\}$	$\{(y,?),(x,1)\}$	$\{(x,1),(y,2),(y,4),(x,5)\}$	$\{(x,1),(y,2),(y,4),(x,5)\}$	$\{(x,1),(x,5),(y,4)\})$

Es gilt $g_S(g_S^4(\bot)) = g_S^4(\bot)$. Also ist $g_S^4(\bot)$ der kleinste Fixpunkt.

Die kleinste Lösung des Gleichungssystems ist

$$X_1 = \{(x,?),(y,?)\} \qquad X_2 = \{(y,?),(x,1)\}$$
$$X_3 = \{(x,1),(y,2),(y,4),(x,5)\} \qquad X_4 = \{(x,1),(y,2),(y,4),(x,5)\}$$
$$X_5 = \{(x,1),(x,5),(y,4)\}.$$

Die kleinste Lösung ist die gewünschte Information. Größere May-Information bedeutet Informationsverlust.

2.5. Beispiel 2: Available-Expressions-Analyse

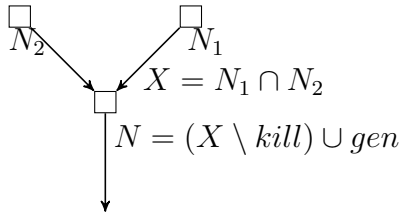
Ziel: Berechne für jeden Block die Ausdrücke, die auf allen Pfaden zu dem Block definitiv berechnet worden sind und nicht zwischendurch geändert wurden.

Klassifikation:

Vorwärtsanalyse, die Information über die Vergangenheit von Daten berechnet.

Must-Analyse, die das gemeinsame Verhalten aller Ausführungen unterapproximiert. Das heißt, die berechnete Information gilt definitiv für alle Ausführungen.

Idee:



Anwendungen: Vermeide erneute Berechnung bekannter Werte.

Beispiel 2.6. Betrachte ein Programm mit Teilausdrücken *AExp* und Blöcken *Blocks*. Nutze $AExp(a)$ für die Teilausdrücke von Ausdruck $a \in AExp$.

Nutze $Vars(a)$ für die Variablen, die in Ausdruck $a \in AExp$ vorkommen.

Definiere das Datenflusssystem

$$S = (G, (D, \preceq), i, \{f_b : D \to D \mid b \in Blocks\}) \ .$$

- Kontrollflussgraph $G = (B, E, F)$:
  $B = Blocks$, $E =$ initialer Block, $F =$ Kontrollfluss in Programmordnung.
- Verband $(D, \preceq)$:
  $(D, \preceq) = (\mathcal{P}(AExp), \supseteq)$.

  Es handelt sich um einen (dualen) Potenzmengenverband. (ACC) gilt, da der Verband endlich ist.
- Anfangswert i:
  $\emptyset$.
- Transferfunktionen $f_b : D \to D$:

$$\begin{aligned} f_b : \mathcal{P}(AExp) &\to \mathcal{P}(AExp) \\ X &\mapsto (X \setminus kill(b)) \cup gen(b) \end{aligned}$$

  Die Mengen $kill(b), gen(b) \subseteq AExp$ sind

$$kill(b) = \begin{cases} \{a' \in AExp \mid x \in Vars(a')\}, & \text{falls } b = [x := a]^b \\ \emptyset, & \text{sonst.} \end{cases}$$

  //Teilausdrücke, die x enthalten und daher von Block b geändert werden.

$$gen(b) = \begin{cases} \{a' \in AExp(a) \mid x \notin Vars(a')\}, & \text{falls } b = [x := a]^b \\ AExp(cond), & \text{falls } b = [cond]^b \\ \emptyset, & \text{sonst.} \end{cases}$$

  //Teilausdrücke, die von Block b genutzt werden.

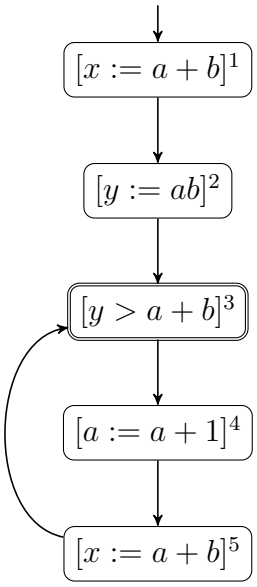
  Beachte, dass bei einer Zuweisung, deren rechte Seite den zugewiesenen Wert beinhaltet, die entsprechenden Ausdrücke nicht available werden, da sich ihr

Wert durch die Zuweisung ändern kann (z.B. ändert sich durch die Zuweisung $a := a + 1$ der Wert von $a + 1$). Daher ist die Einschränkung $x \notin \mathit{Vars}(a')$ oben nötig.

Die Transferfunktionen sind monoton.

Betrachte folgendes Beispielprogramm bzw. den zugehörigen Kontrollflussgraphen.

```
[x:=a+b]1;
[y:=ab]2;
while [y > a+b]3 do
    [a:=a+1]4;
    [x:=a+b]5;
endwhile
```



Die Transferfunktionen sind wie folgt.

Block	$\mathit{kill}(b)$	$\mathit{gen}(b)$	$f_b(X)$
$[x := a + b]^1$	$\emptyset$	$\{a + b\}$	$X \cup \{a + b\}$
$[y := ab]^2$	$\emptyset$	$\{ab\}$	$X \cup \{ab\}$
$[y > a + b]^3$	$\emptyset$	$\{a + b\}$	$X \cup \{a + b\}$
$[a := a + 1]^4$	$\{a + b, ab, a + 1\}$	$\emptyset$	$X \setminus \{a + b, ab, a + 1\}$
$[x := a + b]^5$	$\emptyset$	$\{a + b\}$	$X \cup \{a + b\}$

Das folgende Gleichungssystem wird vom Datenflusssystem induziert.

$$
\begin{aligned}
X_1 &= \underbrace{\emptyset}_{=i} \\
X_2 &= \underbrace{X_1 \cup \{a + b\}}_{=f_1(X_1)} \\
X_3 &= \underbrace{(X_2 \cup \{ab\})}_{=f_2(X_2)} \cap \underbrace{(X_5 \cup \{a + b\})}_{=f_5(X_5)} \\
X_4 &= X_3 \cup \{a + b\} \\
X_5 &= X_4 \setminus \{a + b, ab, a + 1\}
\end{aligned}
$$

Wir berechnen eine Lösung des Gleichungssystems durch Iteration von

$$g_S : \mathcal{P}\,(\mathit{AExp})^5 \to \mathcal{P}\,(\mathit{AExp})^5$$

auf $\bot$ von $(\mathcal{P}(AExp)^5, \supseteq^5)$ bis zum kleinsten Fixpunkt:

Iter.	$(d_1$	d_2	d_3	d_4	$d_5)$
$g_S^0(\bot)$	$(\{a+b, ab, a+1\}$	$\{a+b, ab, a+1\}$	$\{a+b, ab, a+1\}$	$\{a+b, ab, a+1\}$	$\{a+b, ab, a+1\})$
$g_S^1(\bot)$	$(\emptyset$	$\{a+b, ab, a+1\}$	$\{a+b, ab, a+1\}$	$\{a+b, ab, a+1\}$	$\emptyset)$
$g_S^2(\bot)$	$(\emptyset$	$\{a+b\}$	$\{a+b\}$	$\{a+b, ab, a+1\}$	$\emptyset)$
$g_S^3(\bot)$	$(\emptyset$	$\{a+b\}$	$\{a+b\}$	$\{a+b\}$	$\emptyset)$
$g_S^4(\bot)$	$(\emptyset$	$\{a+b\}$	$\{a+b\}$	$\{a+b\}$	$\emptyset)$

Es gilt $g_S(g_S^3(\bot)) = g_S^3(\bot)$. Also ist $g_S^3(\bot)$ der kleinste Fixpunkt.

Die kleinste Lösung des Gleichungssystems ist

$$X_1 = \emptyset = X_5 \qquad\qquad X_2 = \{a+b\} = X_3 = X_4.$$

Die kleinste Lösung ist die gewünschte Information. Größere (bzgl. $\supseteq$) Must-Information bedeutet Informationsverlust.

Bemerkung:
Wir haben hier den größten Fixpunkt auf dem Potenzmengenverband $(\mathcal{P}(AExp), \subseteq)$ berechnet.

Durch Dualisierung des Verbandes zu $(\mathcal{P}(AExp), \supseteq)$ konnten wir eine kleinste Fixpunktberechnung und so unser Framework mit (ACC) nutzen.

2.6. Beispiel 3: Live-Variables-Analyse

Definition: Eine Variable heißt *lebendig* am Ausgang eines Blocks, falls es einen Ablauf von diesem Block zu einem anderen Block geben könnte, der die Variable in einer Bedingung oder auf der rechten Seite einer Zuweisung nutzt, ohne dass die Variable zwischendurch überschrieben wird.

Am Ende des Programms sind alle Variablen lebendig.

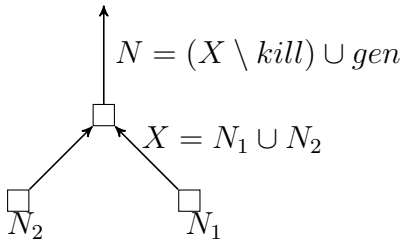
Ziel: Berechne für jeden Block die Variablen, die am Ausgang lebendig sind.

Klassifikation:

Rückwärtsanalyse, die Informationen über die Zukunft von Daten berechnet.

May-Analyse, die das Verhalten aller einzelnen Ausführungen überapproximiert. Das heißt, das Verhalten jeder Ausführung ist sicher in der Information enthalten.

Idee:



Anwendungen:

Register-Allocation: Falls x lebendig ist, wird die Variable vermutlich bald genutzt und sollte ein Register erhalten. Ist x nicht mehr lebendig, kann das Register neu vergeben werden.

Dead-Code-Elimination: Ist x am Ausgang einer Zuweisung (zu x) nicht lebendig, kann die Zuweisung entfernt werden.

Auf ähnliche Weise lassen sich Variablen zusammenfassen: Sind x und y nie gemeinsam lebendig, verwende eine Variable z.

Beispiel 2.7. Betrachte ein Programm mit Variablen Blöcken *Blocks* und Variablen *Vars*.

Ferner sei $Vars(a)$ die Menge der Variablen, die in einem Ausdruck a vorkommen.

Definiere das Datenflusssystem

$$S = (G, (D, \preceq), i, \{f_b : D \to D \mid b \in Blocks\}) \ .$$

- Kontrollflussgraph $G = (B, E, F)$:
  $B = Blocks$, $E =$ finale Blöcke, $F =$ Kontrollfluss *gegen* die Programmordnung.
- Verband $(D, \preceq)$:
  $(D, \preceq) = (\mathcal{P}(Vars), \subseteq)$.

  Es handelt sich um einen Potenzmengenverband. (ACC) gilt, da der Verband endlich ist.
- Anfangswert i:
  $Vars$ (am Ende des Programms sind per Definition alle Variablen lebendig).
- Transferfunktionen $f_b : D \to D$:

$$\begin{aligned} f_b : \mathcal{P}(Vars) &\to \mathcal{P}(Vars) \\ X &\mapsto (X \setminus kill(b)) \cup gen(b) \end{aligned}$$

Die Mengen $kill(b), gen(b) \subseteq Vars$ sind

$$kill(b) = \begin{cases} \{x\}, & \text{falls } b = [x := a]^b \\ \emptyset, & \text{sonst.} \end{cases}$$

//Variablen, die von Block b überschrieben werden.

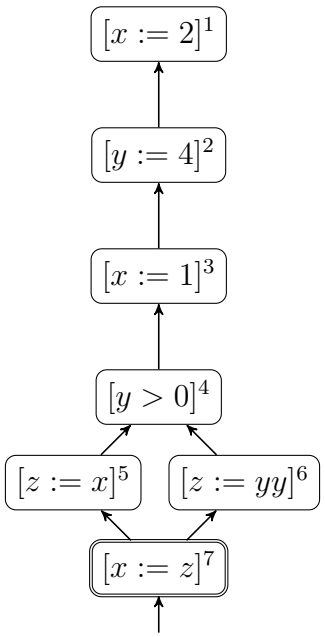
$$gen(b) = \begin{cases} Vars(a), & \text{falls } b = [x := a]^b \\ Vars(cond), & \text{falls } b = [cond]^b \\ \emptyset, & \text{sonst.} \end{cases}$$

//Variablen, die von Block b genutzt werden.

Die Transferfunktionen sind monoton.

Betrachte folgendes Beispielprogramm bzw. den zugehörigen Kontrollflussgraphen.

```
[x:=2]1;
[y:=4]2;
[x:=1]3;
if [y>0]4 then
    [z:=x]5
else
    [z:=yy]6;
endif;
[x:=z]7;
```



Die Transferfunktionen sind wie folgt.

Block	$kill(b)$	$gen(b)$	$f_b(X)$
$[x := 2]^1$	$\{x\}$	$\emptyset$	$X \setminus \{x\}$
$[y := 4]^2$	$\{y\}$	$\emptyset$	$X \setminus \{y\}$
$[x := 1]^3$	$\{x\}$	$\emptyset$	$X \setminus \{x\}$
$[y > 0]^4$	$\emptyset$	$\{y\}$	$X \cup \{y\}$
$[z := x]^5$	$\{z\}$	$\{x\}$	$(X \setminus \{z\}) \cup \{x\}$
$[z := yy]^6$	$\{z\}$	$\{y\}$	$(X \setminus \{z\}) \cup \{y\}$
$[x := z]^7$	$\{x\}$	$\{z\}$	$(X \setminus \{x\}) \cup \{z\}$

Das Datenflusssystem induzierte das folgende Gleichungssystem.

$$
\begin{aligned}
X_1 &= X_2 \setminus \{y\} \\
X_2 &= X_3 \setminus \{x\} \\
X_3 &= X_4 \cup \{y\} \\
X_4 &= \underbrace{((X_5 \setminus \{z\}) \cup \{x\})}_{=f_5(X_5)} \cup \underbrace{((X_6 \setminus \{z\}) \cup \{y\})}_{=f_6(X_6)} \\
X_5 &= (X_7 \setminus \{x\}) \cup \{z\} \\
X_6 &= (X_7 \setminus \{x\}) \cup \{z\} \\
X_7 &= \underbrace{\{x, y, z\}}_{=i}
\end{aligned}
$$

Wir berechnen eine Lösung des Gleichungssystems durch Iteration von

$$g_S : \mathcal{P}\,(\mathit{Vars})^7 \to \mathcal{P}\,(\mathit{Vars})^7$$

auf $\bot$ von $(\mathcal{P}\,(\mathit{Vars})^7, \subseteq^7)$ bis zum kleinsten Fixpunkt:

Iter.	$(d_1$	d_2	d_3	d_4	d_5	d_6	$d_7)$
$g_S^0(\bot)$	$(\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset)$
$g_S^1(\bot)$	$(\emptyset$	$\emptyset$	$\{y\}$	$\{y, x\}$	$\{z\}$	$\{z\}$	$\{x, y, z\})$
$g_S^2(\bot)$	$(\emptyset$	$\{y\}$	$\{y, x\}$	$\{y, x\}$	$\{y, z\}$	$\{y, z\}$	$\{x, y, z\})$
$g_S^3(\bot)$	$(\emptyset$	$\{y\}$	$\{y, x\}$	$\{y, x\}$	$\{y, z\}$	$\{y, z\}$	$\{x, y, z\})$

Es gilt $g_S(g_S^2(\bot)) = g_S^2(\bot)$. Also ist $g_S^2(\bot)$ der kleinste Fixpunkt.

Die kleinste Lösung des Gleichungssystems ist

$$
\begin{aligned}
X_1 &= \emptyset & X_2 &= \{y\} \\
X_3 &= \{y, x\} = X_4 & X_5 &= \{y, z\} = X_6 \\
X_7 &= \{x, y, z\}.
\end{aligned}
$$

Die kleinste Lösung ist die gewünschte Information. Größere May-Information bedeutet Informationsverlust.

Der Block $[x := 2]^1$ kann entfernt werden.

2.7. Beispiel 4: Very-Busy-Expressions-Analyse

Definition: Ein Ausdruck heißt *very busy* am Ausgang eines Blocks, falls der Ausdruck auf jedem Pfad, der von diesem Block ausgeht, verwendet wird, bevor eine der enthaltenen Variablen neu geschrieben wird.

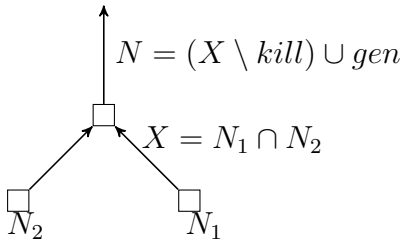
Ziel: Berechne für jeden Block die Ausdrücke, die am Ausgang very busy sind.

Klassifikation:

Rückwärtsanalyse, die Information über die Zukunft von Daten berechnet.

Must-Analyse, die das gemeinsame Verhalten aller Ausführungen unterapproximiert. Das heißt, die berechnete Information gilt definitiv für alle Ausführungen.

Idee:



Anwendungen:

Hoisting-Expressions: Betrachte eine Schleife mit einem Block $x := (a + b)y$, wobei $a + b$ von der Schleife nicht geändert wird. Dann lässt sich eine Zuweisung $t := a + b$ vor der Schleife einfügen und $x := (a + b)y$ durch $x := ty$ ersetzen.

Beispiel 2.8. Betrachte ein Programm mit Teilausdrücken *AExp* und Blöcken *Blocks*.

Nutze $AExp(a)$ für die Teilausdrücke von $a \in AExp$.

Nutze $Vars(a)$ für die Variablen, die in $a \in AExp$ vorkommen.

Definiere das Datenflusssystem

$$S = (G, (D, \preceq), i, \{f_b : D \to D \mid b \in Blocks\}) \ .$$

- Kontrollflussgraph $G = (B, E, F)$:
  $B = Blocks$, $E =$ finale Blöcke, $F =$ Kontrollfluss gegen die Programmordnung.

- Verband $(D, \preceq)$:
  $(D, \preceq) = (\mathcal{P}(AExp), \supseteq)$.

  Es handelt sich um einen dualen Potenzmengenverband. (ACC) gilt, da der Verband endlich ist.

- Anfangswert i:
  $\emptyset$.

- Transferfunktionen $f_b : D \to D$:

$$f_b : \mathcal{P}(AExp) \to \mathcal{P}(AExp)$$
$$X \mapsto (X \setminus kill(b)) \cup gen(b)$$

Die Mengen $kill(b), gen(b) \subseteq AExp$ sind

$$kill(b) = \begin{cases} \{a' \in AExp \mid x \in Vars(a')\}, & \text{falls } b = [x := a]^b \\ \emptyset, & \text{sonst.} \end{cases}$$

//Teilausdrücke, die x enthalten und daher von Block b geändert werden.

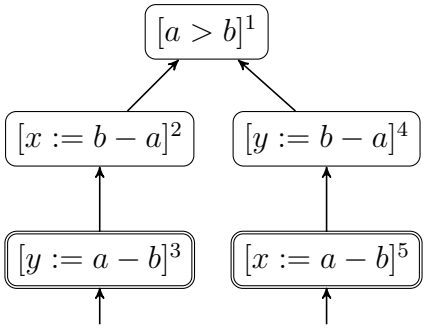
$$gen(b) = \begin{cases} AExp(a), & \text{falls } b = [x := a]^b \\ \emptyset, & \text{sonst.} \end{cases}$$

//Teilausdrücke, die von Block b genutzt werden.

Die Transferfunktionen sind monoton.

Betrachte folgendes Beispielprogramm bzw. den zugehörigen Kontrollflussgraphen.

```
if [a>b]^1 then
    [x:=b-a]^2;
    [y:=a-b]^3
else
    [y:=b-a]^4;
    [x:=a-b]^5
endif
```



Die Transferfunktionen sind wie folgt.

Block	$kill(b)$	$gen(b)$	$f_b(X)$
$[a > b]^1$	$\emptyset$	$\emptyset$	X
$[x := b - a]^2$	$\emptyset$	$\{b - a\}$	$X \cup \{b - a\}$
$[y := a - b]^3$	$\emptyset$	$\{a - b\}$	$X \cup \{a - b\}$
$[y := b - a]^4$	$\emptyset$	$\{b - a\}$	$X \cup \{b - a\}$
$[x := a - b]^5$	$\emptyset$	$\{a - b\}$	$X \cup \{a - b\}$

Das folgende Gleichungssystem wird durch das Datenflusssystem induziert.

$$X_1 = \underbrace{(X_2 \cup \{b - a\})}_{=f_2(X_2)} \cap \underbrace{(X_4 \cup \{b - a\})}_{=f_4(X_4)}$$
$$X_2 = X_3 \cup \{a - b\}$$
$$X_3 = \underbrace{\emptyset}_{=i}$$
$$X_4 = X_5 \cup \{a - b\}$$
$$X_5 = \underbrace{\emptyset}_{=i}$$

Wir berechnen eine Lösung des Gleichungssystems durch Iteration von

$$g_S : \mathcal{P}(AExp)^5 \to \mathcal{P}(AExp)^5$$

auf $\bot$ von $(\mathcal{P}(AExp)^5, \supseteq^5)$ bis zum kleinsten Fixpunkt:

Iter.	$(d_1$	d_2	d_3	d_4	$d_5)$
$g_S^0(\bot)$	$(\{a-b, b-a\}$	$\{a-b, b-a\}$	$\{a-b, b-a\}$	$\{a-b, b-a\}$	$\{a-b, b-a\})$
$g_S^1(\bot)$	$(\{a-b, b-a\}$	$\{a-b, b-a\}$	$\emptyset$	$\{a-b, b-a\}$	$\emptyset)$
$g_S^2(\bot)$	$(\{a-b, b-a\}$	$\{a-b\}$	$\emptyset$	$\{a-b\}$	$\emptyset)$
$g_S^3(\bot)$	$(\{a-b, b-a\}$	$\{a-b\}$	$\emptyset$	$\{a-b\}$	$\emptyset)$

Es gilt $g_S(g_S^2(\bot)) = g_S^2(\bot)$. Also ist $g_S^2(\bot)$ der kleinste Fixpunkt.

Die kleinste Lösung des Gleichungssystems ist

$$X_1 = \{a-b, b-a\} \qquad X_2 = \{a-b\} = X_4 \qquad X_3 = \emptyset = X_5.$$

Die kleinste Lösung ist die gewünschte Information. Größere (bzgl. $\supseteq$) Must-Information bedeutet Informationsverlust.

Bemerkung:
Wir haben hier den größten Fixpunkt auf dem Potenzmengenverband $(\mathcal{P}(AExp), \subseteq)$ berechnet.

Durch Dualisierung des Verbandes zu $(\mathcal{P}(AExp), \supseteq)$ konnten wir eine kleinste Fixpunktberechnung und so unser Framework mit (ACC) nutzen.

Teil II.

Reguläre Sprachen

3. Reguläre Sprachen und endliche Automaten

3.1. Reguläre Sprachen

Grundbegriffe:

- Ein Alphabet ist eine endliche Menge Σ (z.B. $\Sigma = \{a, b, c\}$).
- Ein Wort ist eine endliche Folge $w = a_1 \dots a_n$ mit $a_i \in \Sigma, i = 1, \dots, n \in \mathbb{N}$.
- Die Länge von $w = a_1 \dots a_n$ ist $|w| = n$.
- Für den i-ten Buchstaben schreiben wir $w(i) := a_i$.
- Die Kleenesche Hülle Σ^* ist die Menge aller endlichen Wörter über Σ.
- Die Konkatenation von $w, v \in \Sigma^*$ ist $w.v \in \Sigma^*$. Es gilt $|w.v| = |w| + |v|$.
- Das Wort $\varepsilon \in \Sigma^*$ heißt *leeres Wort* mit $|\varepsilon| = 0$. Das leere Wort ist das neutrale Element der Konkatenation, es gilt also für alle $w \in \Sigma^*$: $w.\varepsilon = \varepsilon.w = w$.
- Eine Sprache ist eine (meistens unendliche) Menge $L \subseteq \Sigma^*$ von Wörtern über einem Alphabet Σ.
- Da Sprachen Mengen sind, greifen mengentheoretische Operationen:
  - Vereinigung: $L_1 \cup L_2$
  - Schnitt: $L_1 \cap L_2$
  - Differenz: $L_1 \setminus L_2$
  - Komplement: $\overline{L_1} := \Sigma^* \setminus L_1$

  All diese Mengen sind ebenfalls Sprachen.
- Die Konkatenation zweier *Sprachen* L_1, L_2 ist die Sprache

$$L_1.L_2 := \{u.v \mid u \in L_1, v \in L_2\} .$$

- Die Sprache

$$L^* := \bigcup_{i \in \mathbb{N}} L^i$$

mit $L^0 := \{\varepsilon\}$, $L^{i+1} := L.L^i$ heißt Kleenesche Hülle von L. Die Operation $\cdot^*$ heißt Kleene-Stern. Die Sprache

$$L^+ := \bigcup_{i \in \mathbb{N}, i>0} L^i$$

heißt positive Hülle von L. Falls L das Wort ε nicht enthält, gilt $L^+ = L^* \setminus \{\varepsilon\}$.

Definition 3.1 (reguläre Sprache)**.** Die Menge der regulären Sprachen über dem Alphabet Σ, notiert als REG_Σ, ist die kleinste Menge, die Folgendes erfüllt:

1) $\emptyset \in \mathrm{REG}_\Sigma$, $\{\varepsilon\} \in \mathrm{REG}_\Sigma$, $\{a\} \in \mathrm{REG}_\Sigma$ $(\forall a \in \Sigma)$

2) $L_1, L_2 \in \mathrm{REG}_\Sigma$ impliziert
$L_1 \cup L_2 \in \mathrm{REG}_\Sigma$,
$L_1.L_2 \in \mathrm{REG}_\Sigma$ und
$L_1{}^* \in \mathrm{REG}_\Sigma$.

Die Klasse der reguläre Sprachen ist die Vereinigung von REG_Σ über alle endlichen Alphabete Σ.

Jede reguläre Sprache wird durch endlich viele Anwendungen der unter 2) aufgeführten Operationen gebildet auf die Basisfälle aus 1).

Notation:

Bindung:	Kleene-Stern > Konkatenation > Vereinigung
Klammern:	schreibe einelementige Singleton-Sprache ohne Mengenklammern, z.B. a statt $\{a\}$.

Beispiel 3.2. Sei $\Sigma = \{a, b\}$.

$(a \cup b)^*.b$	=	Beliebig lange Worte bestehend aus a und b, die auf b enden.
$\Sigma^*.b.a^*.b.\Sigma^*$	=	Worte bestehend aus a und b, die mindestens zwei b's enthalten, zwischen denen beliebig viele a's stehen können.

Nicht regulär: $L = \{a^n b^n \mid n \in \mathbb{N}\}$ (Beweis in Kapitel 7 mit dem Pumping Lemma).

Beobachtungen:

→ Jede endliche Menge von Worten bildet eine reguläre Sprache.

→ Reguläre Sprachen sind nicht unter unendlicher Vereinigung abgeschlossen (betrachte z.B. $\bigcup_{k \in \mathbb{N}} \{a^k b^k\} = \{a^n b^n \in \mathbb{N}\}$).

→ Per Definition ist REG_Σ unter *endlicher* Vereinigung, Konkatenation und Kleene-Stern abgeschlossen.

Ziel: REG_Σ ist auch unter den übrigen mengentheoretischen Operationen, also Schnitt, Differenz und Komplement, abgeschlossen. Um das zu zeigen, geben wir eine alternative Charakterisierung der regulären Sprachen an: endliche Automaten.

3.2. Endliche Automaten

Alphabet Σ sei fest.

Definition 3.3 (NFA)**.** Ein nichtdeterministischer (zustands-)endlicher Automat (nondeterministic finite(-state) automaton – kurz: NFA) über Σ ist ein Tupel

$$A = (Q, q_0, \to, Q_F)$$

mit

- einer endlichen Menge Q von Zuständen,
- dem Startzustand $q_0 \in Q$,
- einer Menge $Q_F \subseteq Q$ von Endzuständen (oder Finalzuständen)
- und der Transitionsrelation $\to \subseteq Q \times \Sigma \times Q$

Für eine Transition $(q, a, q') \in \to$ schreiben wir auch vereinfacht $q \xrightarrow{a} q'$. Ein Ablauf (oder Lauf) von A ist eine Sequenz $q_0 \xrightarrow{a_1} q_1 \ldots \xrightarrow{a_n} q_n$ mit $n \in \mathbb{N}$. Gibt es für ein Wort $w = a_1 ... a_n$ einen Ablauf $q_0 \xrightarrow{a_1} q_1 \ldots \xrightarrow{a_n} q_n$ in A, schreiben wir auch kurz $q_0 \xrightarrow{w} q_n$. Wenn $q_n \in Q_F$, dann heißt dieser Ablauf *akzeptierend*. Die Sprache des Automaten ist $L(A) := \{w \mid A \text{ hat einen akzeptierenden Ablauf auf } w\}$.

Für die grafische Repräsentation von NFAs verwenden wir Graphen mit folgenden Notationselementen:

- Ein Zustand (Knoten des Graphen) wird als Kreis dargestellt, der mit dem Namen des Zustands beschriftet sein kann:

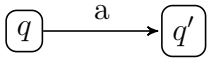
  

  Wenn unerheblich ist, wie genau der Zustand heißt, kann die Beschriftung entfallen.

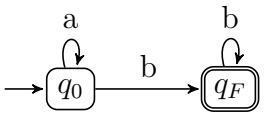
- Finalzustände werden durch Doppelkreise dargestellt und initiale Zustände erhalten einen eingehenden Pfeil:

- Transitionen werden durch gerichtete und beschriftete Kanten dargestellt (hier z.B. (q, a, q')):

Beispiel 3.4. Der folgende Automat akzeptiert $L = \{a^n b^m \mid n, m \in \mathbb{N}\ , m > 0\}$.



Wir wollen zunächst zeigen, dass die Klasse der NFA-Sprachen gleich der Klasse der regulären Sprachen ist.

Satz 3.5 (Kleene)**.** Die Sprachen, die von endlichen Automaten akzeptiert werden, sind genau die regulären Sprachen. Für alle $L \subseteq \Sigma^*$ gilt:

$$L \in \mathrm{REG}_\Sigma \quad \text{gdw.} \quad \exists \text{ NFA } A \text{ mit } L(A) = L \ .$$

Im folgenden beweisen wir beide Richtungen der Äquivalenz.

Lemma 3.6. Sei $a \in \Sigma$. Es gibt NFAs $A_\emptyset, A_\varepsilon, A_a$ mit $L(A_\emptyset) = \emptyset$, $L(A_\varepsilon) = \{\varepsilon\}, A_a = \{a\}$.

Beweis. $A_\emptyset$ ist ein Automat ohne Transitionen und ohne akzeptierende Zustände. A_ε ist der Automat ohne Transitionen, dessen Startzustand akzeptiert. A_a ist der Automat der akzeptiert, sobald er a gelesen hat.

□

Satz 3.7. Seien A und B zwei NFAs. Dann gibt es

i) einen NFA $A.B$ mit $L(A.B) = L(A).L(B)$ sowie

ii) einen NFA $A \cup B$ mit $L(A \cup B) = L(A) \cup L(B)$.

iii) einen NFA A^* mit $L(A^*) = L(A)^*$.

Beweis. Hausaufgabe. □

Satz 3.8. Sei $L \in \mathrm{REG}_\Sigma$. Dann gibt es einen NFA A mit $L = L(A)$.

Beweis. Induktion nach dem Aufbau der regulären Sprachen. Die Basisfälle aus Teil 1) der Definition ($\emptyset$, $\{\varepsilon\}$, $\{a\}$) folgen aus Lemma 3.6. Die induktiven Fälle aus Teil 2) ($L_1 \cup L_2$, $L_1.L_2$, L_1^*) folgen aus Satz 3.7. □

Zeige die Rückrichtung von Kleenes Satz:
Sei A ein NFA, dann ist $L(A) \in \mathrm{REG}_\Sigma$.

Ansatz:

→ Stelle Automaten mit n Zuständen als rekursives Gleichungssystem mit n Gleichungen dar.

→ Berechne eine Lösung mit Ardens Lemma.

Lemma 3.9 (Arden '60)**.** Gegeben seien Sprachen $U, V, L \subseteq \Sigma^*$ mit $\varepsilon \notin U$. Dann gilt:

$$L = U.L \cup V \quad \text{gdw. } L = U^*.V$$

Bemerkung: "⇒" rekursive Gleichung hat eindeutige (geschlossene) Lösung.

Beweis. "⇐": Leicht durch Einsetzen:

$$U.L \cup V = U.(U^*.V) \cup V = U^+.V \cup \varepsilon.V = U^*.V = L\ .$$

"⇒": Sei $L \subseteq \Sigma^*$ mit $L = U.L \cup V$.
Behauptung: $L = U^*.V$.

"⊇": Zeige, dass $U^*.V = (\bigcup_{i\in\mathbb{N}} U^i).V = \bigcup_{i\in\mathbb{N}} (U^i.V) \subseteq L$.
Nutze Induktion nach i und zeige $U^n.V \subseteq L\ \forall n \in \mathbb{N}$.

I.A.: $U^0.V = \varepsilon.V = V \subseteq U.L \cup V = L$.

I.S.: Angenommen $U^n.V \subseteq L$.
Dann ist $U^{n+1}.V = U.(U^n.V) \overset{I.V}{\subseteq} U.L \subseteq U.L \cup V = L$.

Also $U^n.V \subseteq L\ \forall n \in \mathbb{N}$, und damit $\bigcup_{n\in\mathbb{N}} U^n.V \subseteq L$.

"⊆": Angenommen $L \nsubseteq U^*.V$.
Dann gibt es ein kürzestes Wort $w \in L$ mit $w \notin U^*.V$.
Da aber $L = U.L \cup V$, gilt $w \in U.L$ oder $w \in V$.
Es kann nicht $w \in V$ gelten, da $V \subseteq U^*.V$ und so $w \in U^*.V$ ↯.
Also gilt: $w \in U.L$, also $w = u.w'$ mit $u \in U$ und $w' \in L$. Es gilt $u \neq \varepsilon$, da $\varepsilon \notin U$. Also ist w' kürzer als w. Da w das kürzeste Wort in L mit $w \notin U^*.V$ ist und da w' kürzer als w und auch aus L ist, folgt $w' \in U^*.V$.
Da $w' \in U^*.V$ und $u \in U$ gilt: $u.w' \in U^*.V$ ↯. (Widerspruch ($w \notin U^*.V$ und $w \in U^*.V$), also gibt es so ein w nicht).

□

Satz 3.10. Wenn L die Sprache eines NFA ist, dann ist L regulär.

Beweis. Sei $L = L(A)$ mit $A = (Q, q_0, \to, Q_F)$ und $Q = \{q_0, \ldots, q_{n-1}\}$. Definiere für jeden Zustand q_i die Sprache

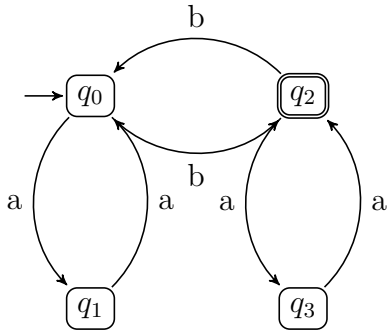
X_i = „Worte, die in A einen akzeptierenden Ablauf von Startzustand q_i haben"
$= L(Q, q_i, \to, Q_F)$

Diese Sprachen erfüllen die Gleichungen

$$X_i = \bigcup_{a\in\Sigma} \bigcup_{q_i \xrightarrow{a} q_j} a.X_j \cup F_i \qquad \text{mit } F_i = \begin{cases} \{\varepsilon\} & \text{falls } q_i \in Q_F\ , \\ \emptyset & \text{sonst}\ . \end{cases}$$

Durch wechselseitiges Einsetzen dieser Gleichungen und Anwendung von Ardens Lemma konstruiere Ausdrücke, die die Regularität der Sprachen X_i nachweisen. Insbesondere ist dann die Regularität von $L = L(A) = X_0$ nachgewiesen. □

Beispiel 3.11. A:



Zugehöriges Gleichungssystem:

$$X_0 = a.X_1 \cup b.X_2 \tag{3.1}$$
$$X_1 = a.X_0 \tag{3.2}$$
$$X_2 = a.X_3 \cup b.X_0 \cup \varepsilon \tag{3.3}$$
$$X_3 = a.X_2 \tag{3.4}$$

Setze (3.4) in (3.3) und (3.2) in (3.1) ein:

$$X_0 = a.a.X_0 \cup b.X_2$$
$$X_2 = a.a.X_2 \cup b.X_0 \cup \varepsilon$$

Mit Ardens Lemma folgt:

$$X_2 = (a.a)^*.(b.X_0 \cup \varepsilon)$$

Einsetzen in X_0 und erneute Anwendung von Ardens Lemma:

$$\begin{aligned} X_0 &= a.a.X_0 \cup b.(a.a)^*.(b.X_0 \cup \varepsilon) \\ &= (a.a \cup b.(a.a)^*.b).X_0 \cup b.(a.a)^* \\ &= (a.a \cup b.(a.a)^*.b)^*.b.(a.a)^* \end{aligned}$$

Satz 3.12 (Allgemeinere Version von Ardens Lemma)**.** Seien $U, V, L \subseteq \Sigma^*$ Sprachen mit $\varepsilon \in U$. Dann gilt:

$$L = U.L \cup V \text{ gdw. } L \in \mathcal{L} = \{U^*.V' \mid V \subseteq V' \subseteq \Sigma^*\}$$

Beweis. Hausaufgabe. □

Definition 3.13 (Produktautomat)**.** Seien $A = (Q^A, q_0^A, \to_A, Q_F^A)$ und $B = (Q^B, q_0^B, \to_B, Q_F^B)$ zwei NFAs. Dann ist der *Produktautomat* $A \times B$ ein NFA, der aus A und B wie folgt hervorgeht:

$$\begin{aligned} A \times B := \quad & (Q^A \times Q^B, (q_0^A, q_0^B), \to_{A\times B}, Q_F^A \times Q_F^B) \\ \text{mit} \quad & (q,p) \xrightarrow{a}_{A\times B} (q',p') \text{ falls } q \xrightarrow{a}_A q' \text{ und } p \xrightarrow{a}_B p' \end{aligned}$$

Satz 3.14. Seien NFAs A und B mit $A = (Q^A, q_0^A, \to_A, Q_F^A)$ und $B = (Q^B, q_0^B, \to_B, Q_F^B)$ gegeben. Der Produktautomat $A \times B$ akzeptiert die Sprache $L(A \times B) = L(A) \cap L(B)$.

Beweis. Hausaufgabe. □

Korollar 3.15. Reguläre Sprachen sind effektiv unter Schnitt abgeschlossen.

Nun: Zeige Abschluss unter Komplement.

Idee: Invertiere Endzustände.

Definition 3.16. Sei $A = (Q, q_0, \to, Q_F)$ ein NFA. Dann ist $\overline{A} := (Q, q_0, \to, Q \setminus Q_F)$ der NFA mit $\overline{L(A)} = L(\overline{A})$.

Problem: $\overline{A}$ akzeptiert nicht das Komplement von $L(A)$.
In Beispiel 3.11: Weder A noch $\overline{A}$ akzeptieren *ab*.

Erklärung:
$L(A)$ = „Worte w, für die es *einen* akzeptierenden Ablauf von A auf w gibt."
$\overline{L(A)}$ = „Worte, so dass *alle* Abläufe von A auf w nicht akzeptierend sind."
$L(\overline{A})$ = „Worte, so dass es *einen* nicht akzeptierenden Ablauf von A auf w gibt"

Man sieht, dass die Sprachen $\overline{L(A)}$ und $L(\overline{A})$ nicht gleich sein müssen.

Lösung: Automatenmodel, bei dem jedes Wort einen eindeutigen Ablauf hat. An diesem Lauf sieht man, ob das Wort in der Sprache ist.

Definition 3.17 (DFA)**.** Ein NFA $A = (Q, q_0, \to, Q_F)$ ist deterministisch falls $\forall q \in Q$ und $\forall a \in \Sigma$ gilt: Es gibt genau ein $q' \in Q$ mit $q \xrightarrow{a} q'$.

Solch ein Automat heißt deterministischer (zustands-)endlicher Automat (DFA - deterministic finite(-state) automaton).

Frage: Gibt es für jeden NFA A einen DFA A' mit $L(A) = L(A')$?.
Antwort: Ja, mittels **Potenzmengenkonstruktion**.

Satz 3.18 (Rabin, Scott '59)**.** Für jeden NFA A mit n Zuständen gibt es einen DFA A' mit $L(A) = L(A')$. Außerdem hat A' höchstens 2^n Zustände.

Beweis. Sei $A = (Q, q_0, \to, Q_F)$. Definiere $A' := (\mathbb{P}(Q), \{q_0\}, \to', Q'_F)$. Der DFA A' enthält für $Q_1 \in \mathbb{P}(Q)$ die Übergänge $Q_1 \xrightarrow{a}{}' Q_2$ mit $Q_2 = \{q_2 \mid \exists q_1 \in Q_1 : q_1 \xrightarrow{a} q_2\}$. Die Finalzustände sind $Q'_F := \{Q' \subseteq Q \mid Q' \cap Q_F \neq \emptyset\}$
Bemerkung: A' ist deterministisch: Für jede Menge Q_1 und jedes a gibt es ein eindeutiges Q_2 (welches auch die leere Menge sein kann, wenn es im NFA keinen Übergang gab). □

Korollar 3.19. Die regulären Sprachen sind genau die Sprachen, die von DFAs akzeptiert werden.

Satz 3.20. Sei A ein DFA. Dann gilt $\overline{L(A)} = L(\overline{A})$.

Beweis. Hausaufgabe. □

Korollar 3.21. Reguläre Sprachen sind effektiv unter Komplement abgeschlossen.

Korollar 3.22. Reguläre Sprachen sind effektiv unter Differenz abgeschlossen.

Beweis. Für $L_1, L_2 \in \mathrm{REG}_\Sigma$ wähle NFA A mit $L_1 = L(A)$ und DFA B mit $L_2 = L(B)$. Dann gilt

$$L_1 \setminus L_2 = L(A) \setminus L(B) = L(A) \cap \overline{L(B)} = L(A) \cap L(\overline{B}) = L(A \times \overline{B}) \ .$$

□

4. ε-Transitionen und Homomorphismen

4.1. Automaten mit ε-Transitionen

Ziel: Erlaube *interne* Transition in Automaten. Diese Transitionen können verwendet werden, ohne dass dabei ein Buchstabe des Eingabeworts verarbeitet wird. Sie erlauben es uns, Automaten kompakter aufzuschreiben.

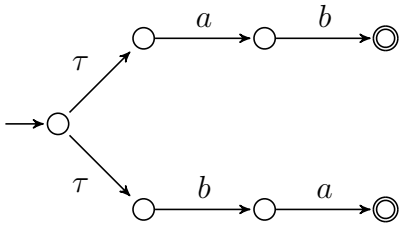
Definition 4.1 (NFA mit internen Transitionen)**.** Ein NFA mit internen Transitionen A besteht aus (τ, A_τ). Hierbei ist $\tau \notin \Sigma$ ein spezielles Symbol für interne Transitionen und $A_\tau = (Q, q_0, \to, Q_F)$ ein NFA über $\Sigma \cup \{\tau\}$.

Die Sprache $L(A)$ ist die Menge der Wörter $w \in \Sigma^*$, so dass es ein $v \in (\Sigma \cup \{\tau\})^*$ gibt mit:

- $v \in L(A_\tau)$ (gewöhnliche Akzeptanz eines NFAs),
- w ergibt sich aus v, indem alle Vorkommen von τ gestrichen werden.

Intuitiv wird ein Wort von A akzeptiert, wenn man durch das Einfügen von τ an geeigneten Stellen ein Wort erzeugen kann, dass von A_τ akzeptiert wird.

Beispiel 4.2. Aus zwei Automaten für $a.b$ und $b.a$ lässt sich der folgende Automat $A = (\tau, A_\tau)$ mit τ-Transitionen für $a.b \cup b.a$ herstellen.



Es ist leicht zu überprüfen, dass $L(A) = ab \cup ba$ gilt. Beispielsweise gibt es zu $w = ab \in \Sigma^*$ das Wort $v = \tau ab \in (\Sigma \cup \{\tau\})^*$ mit $v \in L(A_\tau)$.

Bemerkung. Oft wählt man $\tau = \varepsilon$, beschriftet also interne Transitionen mit ε. Dies ergibt intuitiv Sinn, kann aber zu Verwirrung führen. Wir unterscheiden hier zunächst klar zwischen dem Wort ε und dem Symbol τ.

Wir wollen nun zeigen, dass NFAs mit internen Transitionen ebenfalls genau die regulären Sprachen beschreiben. Eine Richtung des Beweises ist klar: Jede reguläre Sprache ist die Sprache eines NFAs, welchen man als NFA mit internen Transitionen sehen kann, der allerdings gar keine solchen internen Transitionen hat. Die andere Richtung wird durch den folgenden Satz gezeigt.

Satz 4.3. Es sei $A = (\tau, A_\tau)$ ein NFA mit internen Transitionen über Σ. Die Sprache $L(A)$ ist regulär.

Beweis. Es sei $A_\tau = (Q, q_0, \to, Q_F)$ der NFA über $\Sigma \cup \{\tau\}$. Wir konstruieren einen NFA B über Σ mit $L(B) = L(A)$. Intuitiv soll B die τ-Transitionen in A_τ überspringen können.

Wir definieren zu jedem Zustand $q \in Q$ seinen τ-Abschluss $\tau\text{-closure}\,(q)$ als die Menge der Zustände, die sich durch eine Sequenz von τ-Transitionen erreichen lassen,

$$\tau\text{-closure}\,(q) = \{q' \in Q \mid \exists \text{ Transitionssequenz } q \xrightarrow{\tau} \cdots \xrightarrow{\tau} q' \text{ in } A_\tau\} \ .$$

Man beachte, dass $q \in \tau\text{-closure}\,(q)$ gilt, da man die leere Transitionssequenz wählen kann.

Nun definieren wir $B = (Q, q_0, \to', Q'_F)$ mit

- $$Q'_F = \{q \in Q \mid \exists q' \in Q_F \colon q' \in \tau\text{-closure}\,(q)\} \ ,$$
  d.h. ein Zustand q ist Endzustand in B, wenn man von q einen Endzustand von A_τ durch eine Sequenz von τ-Transitionen erreichen kann. (Beachte, dass $Q_F \subseteq Q'_F$ gilt.)
- Für $q_1, q_2 \in Q$, $a \in \Sigma$ gibt es die Transition
  $$q_1 \xrightarrow{a}{}' q_2 \text{ in } B \quad \text{gdw.} \quad \exists q' \in Q \colon q' \in \tau\text{-closure}\,(q_1)\,, q' \xrightarrow{a} q_2 \text{ in } A_\tau \ ,$$
  d.h. wenn es einen Zustand q' gibt, der die Transition $q' \xrightarrow{a} q_2$ in A_τ hat und von q_1 aus durch eine Sequenz von τ-Transitionen erreichbar ist.

Es sei dem Leser als Übung überlassen, zu überprüfen, dass $L(B) = L(A)$ gilt. □

Korollar 4.4. Die Klasse der regulären Sprachen ist gleich der Klasse der Sprachen, die von NFAs mit internen Transition akzeptiert werden.

Bemerkung. Man kann den τ-Abschluss $\tau\text{-closure}\,(q)$ eines Zustands berechnen, in dem man eine Fixpunkt-Konstruktion durchführt. Hierzu sei T_i die Menge der Zustände, die sich von q aus durch höchstens i τ-Transitionen erreichen lassen, induktiv definiert durch $T_0 = \{q\}$, $T_{i+1} = T_i \cup \{q'' \mid \exists q' \in T_i \colon q' \xrightarrow{\tau} q'' \text{ in } A_\tau\}$. Es gilt $T_0 \subseteq T_1 \subseteq T_2 \subseteq \ldots$ Diese aufsteigende Kette wird stationär. Für das kleinste i mit $T_i = T_{i+1}$ gilt $T_i = \tau\text{-closure}\,(q)$.

Bemerkung. Wenn man aus einem gegebenen NFA mit internen Transitionen einen sprachäquivalenten DFA konstruieren möchte, kann man eine Variante der Potenzmengenkonstruktion verwenden, die gleichzeitig die τ-Transition entfernt (statt zuerst die τ-Transitionen zu entfernen und dann die Potenzmengenkonstruktion durchzuführen).

4.2. Homomorphismen

Definition 4.5. Ein Homomorphismus ist eine Funktion $h : \Sigma^* \to \Gamma^*$, so dass $\forall x, y \in \Sigma^*$ gilt:

$$h(x.y) = h(x).h(y)$$

Lemma 4.6. Sei $h : \Sigma^* \to \Gamma^*$ ein Homomorphismus. Es gilt $h(\varepsilon) = \varepsilon$.

Beweis. $\varepsilon = \varepsilon.\varepsilon$, also $h(\varepsilon) = h(\varepsilon.\varepsilon)$ und damit

$$|h(\varepsilon)| = |h(\varepsilon.\varepsilon)| = |h(\varepsilon).h(\varepsilon)| = |h(\varepsilon)| + |h(\varepsilon)| \ .$$

Also $|h(\varepsilon)| = 0$ und damit zwangsweise $h(\varepsilon) = \varepsilon$. □

Satz 4.7. Jede Funktion $f : \Sigma \to \Gamma^*$ lässt sich eindeutig zu einem Homomorphismus $h_f : \Sigma^* \to \Gamma^*$ fortsetzen.

Insbesondere: Jeder Homomorphismus ist eindeutig durch seine Werte auf Σ bestimmt.

Konsequenz: Wir müssen bei einem Homomorphismus nur angeben, wie er auf die einzelnen Buchstaben aus Σ wirkt.

Beweis. Eine Fortsetzung h_f von f, die ein Homomorphismus ist, muss folgende Bedingungen erfüllen:

- $h_f(a) = f(a)$ für $a \in \Sigma$ (da h_f Fortsetzung von f).
- $h_f(\varepsilon) = \varepsilon$ (da h_f Homomorphismus, Lemma 4.6)
- Für Wörter $w = a_1...a_n \in \Sigma^*$ gilt: $h_f(w) = h_f(a_1)...h_f(a_n) = f(a_1)...f(a_n)$, da h Homomorphismus.

Hierdurch ist h_f eindeutig spezifiziert.

Zu einem Homomorphismus $h\colon \Sigma^* \to \Gamma^*$ sei $h_{\restriction\Sigma}\colon \Sigma \to \Gamma^*$ seine Einschränkung auf Σ. Da h eine Fortsetzung von $h_{\restriction\Sigma}$ ist, und diese mit obigen Beweis eindeutig ist, gilt $h = h_{h_{\restriction\Sigma}}$.

Wenn $g\colon \Sigma^* \to \Gamma^*$ ein weiterer Homomorphismus ist, dessen Funktionswerte mit denen von h auf Σ übereinstimmen, also $h_{\restriction\Sigma} = g_{\restriction\Sigma}$, dann gilt

$$g = h_{g_{\restriction\Sigma}} = h_{h_{\restriction\Sigma}} = h \ ,$$

also sind dann g und h bereits gleich. □

Wir wollen nun Homomorphismen auf ganze Sprachen anwenden bzw. rückwärts anwenden.

Definition 4.8. Es sei $h : \Sigma^* \to \Gamma^*$ ein Homomorphismus und $L_1 \subseteq \Sigma^*$ und $L_2 \subseteq \Gamma^*$ Sprachen. Wir definieren das Bild $h(L_1)$ von L_1 unter h und das Urbild $h^{-1}(L_2)$ von L_2 unter h durch

$$h(L_1) := \{h(w) \mid w \in L_1\} \subseteq \Gamma^* ,$$
$$h^{-1}(L_2) := \{w \in \Sigma^* \mid h(w) \in L_2\} \subseteq \Sigma^* .$$

Wir wollen zeigen, dass reguläre Sprachen unter Bildern und Urbildern von Homomorphismen abgeschlossen sind: Wenn L_1 und L_2 regulär sind, dann auch $h(L_1)$ und $h^{-1}(L_2)$.

Proposition 4.9. Es sei $L \subseteq \Sigma^*$ eine reguläre Sprache und $h : \Sigma^* \to \Gamma^*$ ein Homomorphismus. Dann ist $h(L)$ regulär.

Beweis. Angenommen $L = L(A)$ mit A ein NFA über Σ. Wir konstruieren einen NFA $B = (\tau, B_\tau)$ mit internen Transitionen über Γ. Wenn man die internen Transitionen aus diesem Automaten eliminiert (wie im Beweis von Satz 4.3), erhält man einen NFA $h(A)$ mit $h(L(A)) = L(h(A))$.

Wir ersetzen jede mit $x \in \Sigma$ beschriftete Transition $q \xrightarrow{x} q'$ in A durch die Transition $q \xrightarrow{h(x)} q'$ in B. Wenn $h(x) = \varepsilon$ gilt, wird hieraus eine interne τ-Transition. Wenn $h(x)$ aus mehr als einem Buchstaben besteht, dann fügen wir eine Transitionsfolge und die dafür nötigen Zwischenzustände zu B hinzu.

Behauptung: $h(L(A)) = L(B)$.

"$\subseteq$": Sei $w \in h(L(A))$, also $w = h(x)$ für ein $x \in L(A)$.
Es sei

$$q_0 \xrightarrow{x_1} q_1 \xrightarrow{x_2} \ldots \xrightarrow{x_n} q_n$$

ein akzeptierender Lauf von A zu $x = x_1 \ldots x_n$. Zu jedem x_i definieren wir v_i durch $v_i = h(x_i)$ falls $h(x_i) \neq \varepsilon$, $v_i = \tau$ sonst. Gemäß der Konstruktion von B_τ ist dann

$$q_0 \xrightarrow{v_1} q_1 \xrightarrow{v_2} \ldots \xrightarrow{v_n} q_n$$

ein akzeptierender Lauf von B_τ zu $v_1 \ldots v_n \in (\Sigma \cup \{\tau\})^*$. Man beachte, dass hierbei $q_i \xrightarrow{v_{i+1}} q_{i+1}$ eine Transitionsfolge (statt einer einzelnen Transition) bezeichnet, falls $|v_i| > 1$. Durch das Streichen der Vorkommen von τ aus $v_1 \ldots v_n$ erhalten wir w, und es gilt $h(x) = w \in L(B)$ wie gewünscht.

"$\supseteq$": Sei $w \in L(B)$. Dann gibt es ein $v \in (\Sigma \cup \{\tau\})^*$ mit $v \in L(B_\tau)$, so dass w aus v durch das Streichen von τ's entsteht. Es sei

$$q_0 \xrightarrow{v_1} q_1 \xrightarrow{v_2} \ldots \xrightarrow{v_n} q_n$$

ein akzeptierender Ablauf von B_τ auf $v = v_1 \ldots v_n$. Gemäß der Konstruktion von B_τ ist dieser Lauf von der Form

$$q_0 \xrightarrow{h(x_1)} q_{i_1} \xrightarrow{h(x_2)} \ldots \xrightarrow{h(u_k)} q_{i_k} = q_n$$

für geeignet gewählte $x_1, \ldots, x_k \in \Sigma^*$. Gemäß der Konstruktion von B_τ ist

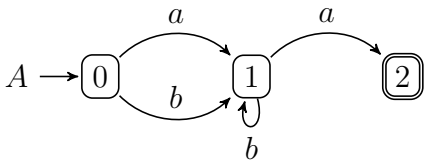
$$q_0 \xrightarrow{x_1} q_{i_1} \xrightarrow{x_2} \ldots \xrightarrow{x_k} q_{i_k} = q_n$$

ein akzeptierender Lauf von A zu $x_1 \dots x_k$. Es gilt also $x_1 \dots x_k \in L(A)$ und damit $h(x_1 \dots x_k) = w \in h(L(A))$ wie gewünscht.

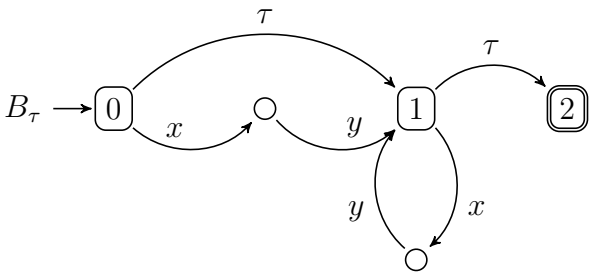
□

Beispiel 4.10. Es sei $\Sigma = \{a, b\}$, $\Gamma = \{x, y, z\}$ und $h\colon \Sigma^* \to \Gamma^*$ der (eindeutige) Homomorphismus mit $h(a) = \varepsilon, h(b) = xy$.

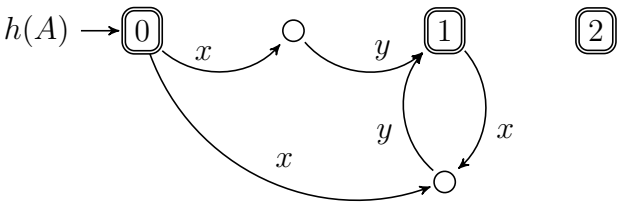
Betrachte den Automaten A über Σ:



Wir erhalten den Automaten $B = (\tau, B_\tau)$:



Durch Eliminieren der τ-Transitionen erhalten wir $h(A)$:



Proposition 4.11. Es sei $L \subseteq \Gamma^*$ eine reguläre Sprache und $h : \Sigma^* \to \Gamma^*$ ein Homomorphismus. Dann ist $h^{-1}(L)$ regulär.

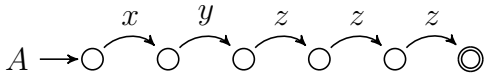
Beweis. Angenommen $L = L(A)$ mit $A = (Q, q_0, \to, Q_F)$ ein NFA über Γ. Wir konstruieren einen NFA $h^{-1}(A) = (Q, q_0, \to', Q_F)$ über Σ wie folgt: Es gibt eine Transition $q \xrightarrow{a}{}' q'$ in $h^{-1}(A)$, wenn es in A eine Transitionsfolge $q \xrightarrow{w_1} q_1 \xrightarrow{w_2} \dots \xrightarrow{w_n} q_n = q'$ gibt mit $h(a) = w_1 \dots w_2$.

Im Spezialfall $h(a) = \varepsilon$ bedeutet dies, dass es für jeden Zustand q eine Schleife $q \xrightarrow{a}{}' q$ gibt.

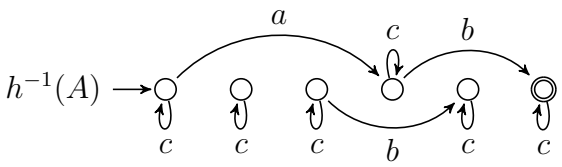
Man kann nun wie in Proposition 4.9 zeigen, dass $L(h^{-1}(B)) = h^{-1}(L(B))$ gilt. □

Beispiel 4.12. Es sei $\Sigma = \{a, b, c\}$, $\Gamma = \{x, y, z\}$ und $h\colon \Sigma^* \to \Gamma^*$ der (eindeutige) Homomorphismus mit $h(a) = xyz, h(b) = zz, h(c) = \varepsilon$.

Es sei A:



Es folgt $h^{-1}(A)$:



Korollar 4.13. Die Klasse der regulären Sprachen ist unter Bildern und Urbildern von Homomorphismen abgeschlossen.

Dieses Ergebnis ermöglicht es uns, leichter zu zeigen, dass Sprachen regulär bzw. nicht-regulär sind.

Beispiel 4.14. Betrachte $\Sigma = \{a, b, c, d, e\}$. Zu einem Wort w und einem Buchstaben x sei $|w|_x$ die Anzahl der Vorkommen von x in w.

a) Betrachte

$$L = \{w \in \Sigma^* \mid |w|_a + |w|_b + |w|_c \equiv |w|_d + |w|_e \text{ modulo } 3\} \ .$$

Um zu zeigen, dass L regulär ist, reicht es zu zeigen, dass

$$L' = \{w \in \{0,1\}^* \mid |w|_0 \equiv |w|_1 \text{ modulo } 3\}$$

regulär ist, denn es gilt $L = h^{-1}(L')$ für den Homomorphismus h mit $h(a) = h(b) = h(c) = 0$ und $h(d) = h(e) = 1$.

Die Sprache L' ist regulär, denn

$$\begin{aligned} L' = \quad & \{w \in \{0,1\}^* \mid |w|_0 \equiv |w|_1 \equiv 0 \bmod 3\} \\ & \cup \{w \in \{0,1\}^* \mid |w|_0 \equiv |w|_1 \equiv 1 \bmod 3\} \\ & \cup \{w \in \{0,1\}^* \mid |w|_0 \equiv |w|_1 \equiv 2 \bmod 3\} \\ = \quad & (\{w \in \{0,1\}^* \mid |w|_0 \equiv 0 \bmod 3\} \cap \{w \in \{0,1\}^* \mid |w|_1 \equiv 0 \bmod 3\}) \\ & \cup (\{w \in \{0,1\}^* \mid |w|_0 \equiv 1 \bmod 3\} \cap \{w \in \{0,1\}^* \mid |w|_1 \equiv 1 \bmod 3\}) \\ & \cup (\{w \in \{0,1\}^* \mid |w|_0 \equiv 2 \bmod 3\} \cap \{w \in \{0,1\}^* \mid |w|_1 \equiv 2 \bmod 3\}) \ . \end{aligned}$$

Es sei der Leserin/dem Leser als Übungsaufgabe überlassen, zu Beweisen, dass die Sprachen von der Form $\{w \in \{0,1\}^* \mid |w|_i \equiv j \bmod 3\}$ regulär sind.

b) Betrachte

$$L = \{w \in \Sigma^* \mid |w|_a + |w|_b + |w|_c = |w|_d + |w|_e\} .$$

Die Sprache L ist nicht regulär: Angenommen sie wäre es. Definiere den Homomorphismus $h\colon \Sigma^* \to \{0,1\}$ durch mit $h(a) = h(b) = h(c) = 0$ und $h(d) = h(e) = 1$. Dann wäre auch

$$h(L) = \{w \in \{0,1\}^* \mid |w|_0 = |w|_1\}$$

regulär und damit auch

$$h(L) \cap 0^*1^* = \{0^n1^n \mid n \in \mathbb{N}\} .$$

Wir wissen aber bereits (und werden in Kapitel 7 formal beweisen), dass diese Sprache nicht regulär ist.

5. Entscheidbarkeit und Komplexität

Die Theoretische Informatik befasst sich insbesondere mit Fragen der *Berechenbarkeit* und *Komplexität*, versucht also zu verstehen, welche Probleme von Computern prinzipiell gelöst werden können. Für die Probleme, die von Computern gelöst werden können, möchte man verstehen, welche Ressourcen (d.h. wie viel Rechenzeit, wie viel Speicherplatz) das Lösen der Probleme benötigt. Im Teilgebiet der *Verifikation* interessiert man sich für Berechnungsprobleme, bei denen Computer bzw. Programme Teil der Eingabe sind. Man interessiert sich also dafür, was Computer über Computer berechnen können.

Im folgenden betrachten wir Berechnungsprobleme, bei denen endliche Automaten (welche eine sehr einfache Form von Computern sind) Teil der Eingabe sind.

Wortproblem für reguläre Sprachen (WORDREG)	
Gegeben:	NFA A über Σ, Wort $w \in \Sigma^*$.
Frage:	Gilt $w \in L(A)$?

Satz 5.1. WORDREG ist entscheidbar und kann in $O(|w| \cdot |Q^2|)$ Zeit gelöst werden.

Man könnte den gegebenen Automaten A zu A^{det} determinisieren und dann den Lauf von A^{det} zu Wort w bestimmen. Da Determinisierung sehr aufwendig ist, erhalten wir dann allerdings nicht die gewünschte Laufzeit. Im Folgenden präsentieren wir ein effizienteres Verfahren.

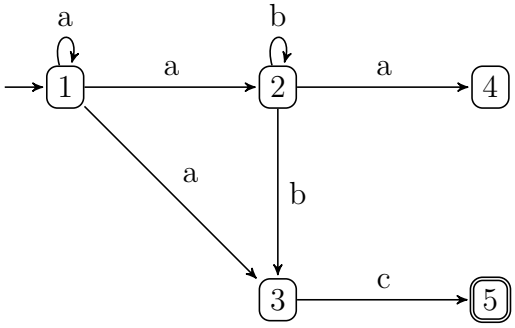
Beweis. Führe eine Potenzmengenkonstruktion *entlang des gegebenen Wortes* durch. Wir bestimmen also zu jedem Präfix $w_1 \dots w_i$ des gegeben Worts w den Zustand Q_i, in dem A^{det} nach dem Lesen des Präfixes wäre.

Es sei A der gegebene Automat und $w = w_1 \dots w_n$ das gegebene Wort.

- Setze $Q_0 = \{q_0\}$ // Startzustand von A^{det}
- Für $i = 1, 2, \dots, n$:
  - Berechne Q_i mit $Q_{i-1} \xrightarrow{w_i} Q_i$ in A^{det},
    also $Q_i = \{q \mid \exists q' \in Q_{i-1} \colon q' \xrightarrow{w_i} q\}$
- Wenn Q_n einen Endzustand enthält, gebe „ja“ zurück, ansonsten „nein“

Die Schleife erfordert $|w|$ Durchläufe. In jedem Durchlauf müssen wir für höchstens $|Q|$ Zustände prüfen, ob es einen Übergang zu jedem der $|Q|$ Zustände gibt. □

Beispiel 5.2. A:



$w = abc$

$Q_0 = \{1\} \xrightarrow{a} Q_1 = \{1, 2, 3\} \xrightarrow{b} Q_2 = \{2, 3\} \xrightarrow{c} Q_3 = \{5\}$

Q_3 enthält Endzustand 5

$\Rightarrow$ Rückgabe „ja“

Leerheitsproblem für reguläre Sprachen (EMPTYREG)	
Gegeben:	NFA A über Σ.
Frage:	Gilt $L(A) = \emptyset$?

Satz 5.3. EMPTYREGist entscheidbar und kann in $O(| \to |)$ Zeit gelöst werden.

Idee: Die Sprache von $L(A)$ ist nicht-leer, wenn es in A einen Endzustand gibt, der vom Startzustand aus erreichbar ist. Wir berechnen also die Menge aller vom Startzustand aus erreichbaren Zustände und überprüfen, ob diese Menge einen Endzustand enthält.

Formal definieren wir eine aufsteigende Kette $R_0 \subseteq R_1 \subseteq R_2 \subseteq ...$ von Mengen von Zuständen:

$$R_i := \text{Zustände, die in } \leq i \text{ Schritten erreichbar sind}$$

Die Berechnung bricht ab, sobald $R_k = R_{k+1}$. In diesem Fall ist ein Fixpunkt (der zu Grunde liegenden Funktion) erreicht,

$$\bigcup_{i \in \mathbb{N}} R_i = R_0 \cup R_1 \cup ... \cup R_k \cup R_{k+1} \cup ... = R_k \ .$$

Beweis. Sei $A = (Q, q_0, \to, Q_F)$. Setze $R_0 := \{q_0\}$ und iteriere mit

$$R_{i+1} := R_i \cup \{q' \in Q \mid \exists q \in R_i : \exists a \in \Sigma : q \xrightarrow{a} q'\}$$

Sei k so, dass $R_k = R_{k+1}$ gilt. Wenn $R_k \cap Q_F = \emptyset$ gebe „ja“ aus, ansonsten „nein“. Im Worst-Case gilt $k = |Q|$. Die Fixpunktiteration benötigt also höchstens $|Q|$ Schritte und der gesamte Algorithmus hat also auf den ersten Blick $O(|Q| \cdot | \to |)$ Zeit. Durch das Verwenden geeigneter Datenstrukturen kann man Erreichen, dass jede Transition nur höchstens einmal betrachtet wird, wodurch man die gewünschte Laufzeit von $O(| \to |)$ erhält. □

Es gibt auch schwerere Probleme, für die bislang kein effizienter Algorithmus bekannt ist (und man vermutet, dass es keinen solchen Algorithmus gibt). Effizient bedeutet hierbei, dass der Zeitverbrauch des Algorithmus durch ein Polynom in der Eingabegröße beschränkt ist.

Universalitätsproblem für reguläre Sprachen (UNIVERSALITYREG)
Gegeben: NFA A über Σ.
Frage: Gilt $L(A) = \Sigma^*$?

Reguläre Inklusion (INCLUSIONREG)
Gegeben: NFAs A, B über Σ.
Frage: Gilt $L(A) \subseteq L(B)$?

Reguläre Äquivalenz (EQUIVALENCEREG)
Gegeben: NFAs A, B über Σ.
Frage: Gilt $L(A) = L(B)$?

Produktleerheit regulärer Sprachen (PRODUCTEMPTYREG)
Gegeben: NFAs $A_1, A_2, A_3, \ldots A_n$ über Σ.
Frage: Gilt $\bigcap_{i \in \{1,\ldots,n\}} L(A_i) = \emptyset$?

Satz 5.4. Die Probleme UNIVERSALITYREG, INCLUSIONREG, EQUIVALENCEREG und PRODUCTEMPTYREG sind entscheidbar und lassen sich in exponentieller Zeit lösen.

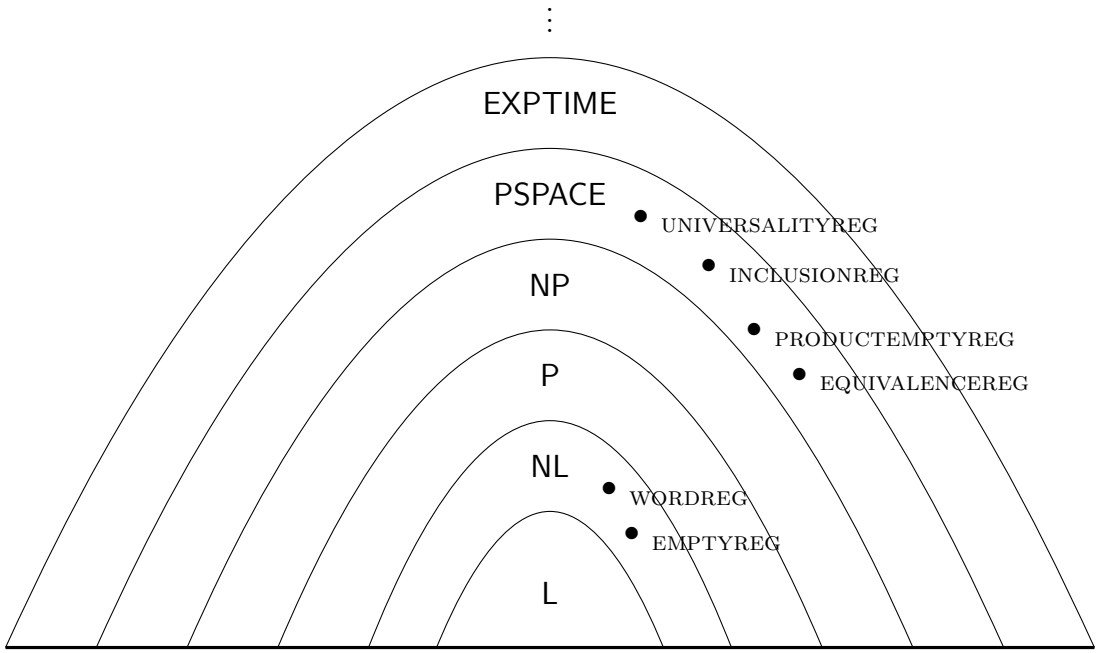
Beweis. Übung. □

Bemerkung. Das Lösen der meisten genannten Probleme wird einfacher, wenn man annimmt, dass die reguläre Sprache nicht als NFA, sondern als DFA, gegeben ist.

Dies ist besonders drastisch im Fall von INCLUSIONREG, EQUIVALENCEREG und UNIVERSALITYREG. Für diese Probleme gibt es keinen bekannten Algorithmus, der die Probleme in polynomieller Zeit löst. Wenn man annimmt, dass die Eingaben DFAs sind, sind diese Probleme jedoch in Polynomialzeit lösbar.

Beispielsweise kann man leicht einen Algorithmus angegeben, der für einen DFA A in Zeit $O(|Q| \cdot | \to |)$ entscheidet, ob $L(A) = \Sigma^*$ gilt.

Bemerkung. Um die Schwierigkeit von Problemen zu charakterisieren, definiert man in der Komplexitätstheorie eine aufsteigende Kette von Komplexitätsklassen. Intuitiv beinhalten "höhere" (größere) Komplexitätsklassen schwierigere Probleme.



Eine formale Definition der Klassen wird in "Theoretische Informatik 2" gegeben. Wichtig ist hier nur:

- Alle Probleme in den Klassen L, NL und P lassen sich in polynomieller Zeit lösen.
- Die Klasse $\mathsf{EXPTIME}$ (und die höheren Klassen) beinhalten Probleme, für die bewiesen ist, dass sie sich nicht in polynomieller Zeit lösen lassen.
- Die Klassen dazwischen, wie NP und PSPACE, beinhalten Probleme, für die kein Algorithmus bekannt ist, der die Probleme in Polynomialzeit löst. Man glaubt, dass auch diese Klassen Probleme beinhalten, die sich nicht in Polynomialzeit lösen lassen (also $\mathsf{NP} \neq \mathsf{P}$ gilt), dies ist allerdings nicht bewiesen.

Für jedes der zuvor genannten Probleme geben wir nun die "niedrigste" (kleinste) Komplexitätsklasse, von der bekannt ist, dass das Problem sich in ihr befindet, an:

- WORDREG ist in NL.
- WORDREG ist in L, wenn wir uns auf DFAs als Eingabe beschränken.
- EMPTYREG ist in NL.
- UNIVERSALITYREG, INCLUSIONREG, EQUIVALENCEREG, *und* PRODUCTEMPTYREG sind in PSPACE.

6. Minimierung

Wir haben im vorherigen Kapitel gesehen, dass wir mit endlichen Automaten Entscheidungsprobleme für unendlich große Sprachen lösen können. Da die Zeitkomplexität unserer Lösungsalgorithmen von der Automatengröße abhängt, sind wir daran interessiert, Automaten möglichst klein zu halten. Um dies zu bewerkstelligen, wollen wir reguläre Ausdrücke nicht direkt in NFAs/DFAs übersetzen, sondern vielmehr die Anforderung einer Sprache an einen Automaten verstehen. Diese Anforderung werden wir dann in einen *kleineren* Automaten übersetzen.

6.1. Der Satz von Myhill-Nerode

Wir beginnen mit der Herleitung einer *notwendigen* und *hinreichenden* Bedingung, einer Charakterisierung, für die Regularität einer Sprache.

Bemerkung. Eigenschaften von Äquivalenzen und Kongruenzen. Eine Relation $\equiv$ heiß Äquivalenzrelation, wenn sie reflexiv, symmetrisch und transitiv ist. Hier betrachten wir Äquivalenzen auf Wörtern, also $\equiv \subseteq \Sigma^* \times \Sigma^*$. Wie üblich schreiben wir $u \equiv v$ statt $(u, v) \in \equiv$.

Eine Äquivalenzrelation heißt Kongruenz, wenn Sie mit einer Operation verträglich ist. In unserem Fall ist diese Operation die Konkatenation von Wörtern. Eine Rechtskongruenz soll

$$u \equiv v \Rightarrow \forall w \in \Sigma^* \colon u.w \equiv v.w$$

erfüllen (für alle $u, v \in \Sigma^*$).

Sei $u \in \Sigma^*$ ein Wort und $\equiv \subseteq \Sigma^* \times \Sigma^*$ eine Äquivalenzrelation. Die *Äquivalenzklasse* von u ist die Menge der zu u äquivalenten Wörter:

$$[u]_{\equiv} = \{v \in \Sigma^* \mid v \equiv u\}.$$

Es gilt:

- Jedes Wort ist in seiner eigenen Äquivalenzklasse enthalten: $u \in [u]_{rel}$.
- Die Äquivalenzklassen von äquivalenten Wörtern sind gleich: $u \equiv v \Rightarrow [u]_{rel} = [v]_{rel}$.
- Die Äquivalenzklassen von nicht-äquivalenten Wörtern sind disjunkt: $u \not\equiv v \Rightarrow [u]_{rel} \cap [v]_{rel} = \emptyset$.
- Die Menge aller Wörter kann disjunkt in Äquivalenzklassen zerlegt werden: $\Sigma^* = \dot{\bigcup}_{u \in \Sigma^*} [u]_{\equiv}$.

Definition 6.1. (Nerode-Rechtskongruenz) Sei $L \subseteq \Sigma^*$ eine Sprache. Die Nerode-Rechtskongruenz $\equiv_L$ ist eine Relation über Σ^*, also eine Teilmenge $\equiv_L \subseteq \Sigma^* \times \Sigma^*$. Sie ist definiert durch:

$$u \equiv_L v, \text{ falls für alle } w \in \Sigma^* \text{ gilt: } u.w \in L \Leftrightarrow v.w \in L.$$

Wir identifizieren also $u \equiv_L v$ (u *kongruent* zu v), genau dann, wenn sich u und v bezüglich Zugehörigkeit zu L gleich verhalten, unabhängig davon, welches Wort man anhängt: Entweder ($u.w \in L$ und $v.w \in L$) oder ($u.w \notin L$ und $v.w \notin L$).

Lemma 6.2. Die Nerode-Rechtskongruenz ist eine Rechtskongruenz und insbesondere eine Äquivalenzrelation.

Bemerkung. Für u aus L gilt: $[u]_{\equiv_L} \subseteq L$.

Definition 6.3. Die Anzahl der Äquivalenzklassen zu einer Äquivalenzrelation heißt *Index* der Äquivalenzrelation.

Beispiel 6.4. Sei L die Sprache definiert durch $L = \{a^n b^n \mid n \in \mathbb{N}\}$. Es gelten folgende Beziehungen: $a \not\equiv_L aab$, da $a.\underline{abb} \in L$, aber $aab.\underline{abb} \notin L$, und $aab \equiv_L aaabb$. Dies kann verallgemeinert werden zu:

- $[a^k]_{\equiv_L} = \{a^k\}$ und
- $[a^{k+1}b]_{\equiv_L} = \{a^{l+1}b^{l+1-k} \mid l \geq k\}$

für alle $k \in \mathbb{N}$. Man erkennt hier schon, dass L unendlichen Index hat.

Satz 6.5 (Myhill-Nerode, 1957 und 1958)**.** Eine Sprache $L \subseteq \Sigma^*$ ist regulär genau dann, wenn $\equiv_L$ endlichen Index hat.

Beweis. Sei L zunächst regulär. Dann gibt es einen DFA $A = (Q, q_0, \to, Q_F)$ mit der Eigenschaft $L(A) = L$. Betrachte nun die Relation $\equiv_A \subseteq \Sigma^* \times \Sigma^*$, definiert durch: $u \equiv_A v$, falls $\exists q \in Q : q_0 \xrightarrow{u} q$ und $q_0 \xrightarrow{v} q$. Dann ist $\equiv_A$ eine Äquivalenzrelation über Σ^* (Beweis: Übung).

Zudem ist $\equiv_A$ eine Teilrelation von $\equiv_L$, also $\equiv_A \subseteq \equiv_L$. Denn für $u, v \in \Sigma^*$ mit $u \equiv_A v$ gilt, für ein beliebiges $w \in \Sigma^*$:

$$\begin{aligned} u.w \in L = L(A) &\Leftrightarrow \exists q \in Q, q_f \in Q_F : q_0 \xrightarrow{u} q \xrightarrow{w} q_f \\ &\Leftrightarrow \exists q \in Q, q_f \in Q_F : q_0 \xrightarrow{v} q \xrightarrow{w} q_f \\ &\Leftrightarrow v.w \in L(A) = L. \end{aligned}$$

Insgesamt also: $u \equiv_L v$. Man beachte, dass die Äquivalenz in Zeile zwei nur gilt, da A ein DFA ist und $u \equiv_A v$. Da $\equiv_A \subseteq \equiv_L$, gilt auch: Index($\equiv_L$) $\leq$ Index($\equiv_A$) und nach Definition von $\equiv_A$ ist Index($\equiv_A$) $\leq |Q|$.

Sei nun der Index von $\equiv_L$ endlich. Wir konstruieren den Äquivalenzklassen-Automaten für L. Dessen Zustände speichern, welche Äquivalenzklasse man

erreicht hat. Es sei $k = \text{Index}(\equiv_L)$ und $u_1, \ldots, u_k \in \Sigma^*$ Repräsentanten der k Äquivalenzklassen. Nach den Eigenschaften für Äquivalenzrelationen gilt: $\Sigma^* = [u_1]_{\equiv_L} \cup \cdots \cup [u_k]_{\equiv_L}$ und auch $[u_i]_{\equiv_L} \cap [u_j]_{\equiv_L} = \emptyset$, für alle $i \neq j$. Also gibt es für jedes $w \in \Sigma^*$ genau ein u_i mit $w \in [u_i]_{\equiv_L}$.
Wir definieren den Äquivalenzklassen-Automaten wie folgt: $A_L = (Q_L, q_{0L}, \rightarrow_L, Q_{FL})$, wobei $Q_L = \{[u_1]_{\equiv_L}, ..., [u_k]_{\equiv_L}\}$, $q_{0L} = [\varepsilon]_{\equiv_L}$, $Q_{FL} = \{[u_j]_{\equiv_L} \mid u_j \in L\}$ und die Transitionen gegeben sind durch: $[u_i]_{\equiv_L} \xrightarrow{a}_L [u_i.a]_{\equiv_L}$ für alle $1 \leq i \leq k$ und $a \in \Sigma$.
Nach Konstruktion gilt für jedes $w \in \Sigma^*$:

$$[\varepsilon]_{\equiv_L} \xrightarrow{w}_L [w]_{\equiv_L}.$$

Daraus erhalten wir: $L(A_L) = L$, denn für $w \in \Sigma^*$ gilt:

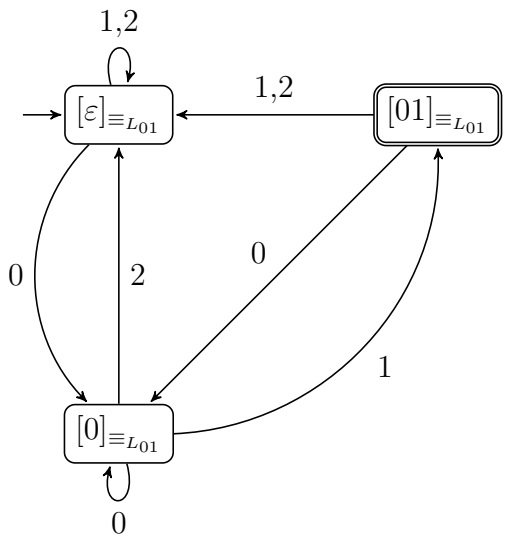
$$\begin{aligned} w \in L(A_L) &\Leftrightarrow \exists [u_j]_{\equiv_L} \in Q_{FL} : [\varepsilon]_{\equiv_L} \xrightarrow{w}_L [u_j]_{\equiv_L} \\ &\Leftrightarrow \exists u_j \in L : [u_j]_{\equiv_L} = [w]_{\equiv_L} \\ &\Leftrightarrow w \in L. \end{aligned}$$

Beachte, dass die Äquivalenz in Zeile zwei nur gilt, da A_L ein DFA ist. □

Beispiel 6.6 (Suffix Detection)**.** Betrachte die Sprache $L_{01} = \{w.01 \mid w \in \Sigma^*\}$ über dem Alphabet $\Sigma = \{0, 1, 2\}$. Wir bestimmen den Index von $\equiv_{L_{01}}$:

- $[0]_{\equiv_{L_{01}}}$ besteht aus allen Wörtern, die auf 0 enden,
- $[01]_{\equiv_{L_{01}}}$ umfasst alle Wörter, die auf 01 enden und
- $[\varepsilon]_{\equiv_{L_{01}}}$ besteht aus allen übrigen Wörtern.

Der Index von $\equiv_{L_{01}}$ ist daher 3. Der Äquivalenzklassen-Automat ist gegeben durch $A_{L_{01}}$:



6.2. Minimale deterministische Automaten

Aufbauend auf dem Satz von Myhill-Nerode erhalten wir das folgende Resultat:

Satz 6.7. Sei L regulär. Dann gibt es einen eindeutigen minimalen DFA A_L mit $L(A_L) = L$, der Index($\equiv_L$) viele Zustände hat.

Der Beweis des Satzes gliedert sich in drei Schritte.

- **Existenz:** Es gibt einen DFA für L mit Index($\equiv_L$) vielen Zuständen.
- **Minimalität:** Es gibt keinen DFA für L mit Zustandsanzahl echt kleiner als Index($\equiv_L$).
- **Eindeutigkeit:** Es gibt keine zwei verschiedene DFAs für L mit genau Index($\equiv_L$) vielen Zuständen.

Dass wir Satz den minimalen Automaten A_L genannt haben ist kein Zufall. Der Äquivalenzklassen-DFA aus dem Beweis von Myhill-Nerode ist *der* minimale DFA für L.

Existenz:

Lemma 6.8. Sei L regulär. Dann gibt es einen DFA für L mit Index($\equiv_L$) vielen Zuständen.

Beweis. Wähle A_L als den Äquivalenzklassenautomaten für L. Wie wir in der Rückrichtung des Beweises des Satzes von Myhill-Nerode gesehen haben, erfüllt A_L die Anforderungen. □

Minimalität:

Lemma 6.9. Sei L regulär. Es gibt keinen DFA für L mit Zustandsanzahl echt kleiner als Index($\equiv_L$).

Beweis. Betrachte einen beliebigen DFA $B = (Q_B, q_{0B}, \rightarrow_B, Q_{FB})$ mit $L(B) = L$. Aus dem Beweis des Satzes von Myhill-Nerode können wir entnehmen, dass

$$\text{Index}(\equiv_L) \leq \text{Index}(\equiv_B) \leq |Q_B|$$

gilt. □

Eindeutigkeit: Es verbleibt zu zeigen, dass es keinen weiteren Automaten A mit $L(A) = L$ und Zustandsanzahl Index($\equiv_L$) gibt. Jeder Automat mit dieser Eigenschaft ist eine Kopie von A_L.

Der folgende Begriff der *Isomorphie* formalisiert dies.

Definition 6.10 (Isomorphie zwischen Automaten)**.** Es seien $A_1 = (Q_1, q_{01}, \to_1, Q_{F1})$ und $A_2 = (Q_2, q_{02}, \to_2, Q_{F2})$ zwei NFAs. Diese sind *isomorph*, wenn es eine Bijektion $\beta : Q_1 \to Q_2$ gibt, mit:

- $\beta(q_{01}) = q_{02}$,
- $\beta(Q_{F1}) = Q_{F2}$ und
- für alle $q, q' \in Q_1$ und $a \in \Sigma$ gilt: $q \xrightarrow{a}_1 q' \Leftrightarrow \beta(q) \xrightarrow{a}_2 \beta(q')$.

Die Abbildung β nennt man Isomorphismus zwischen A_1 und A_2.

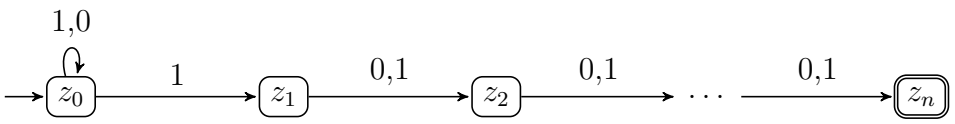
Bemerkung. Isomorphismen definieren eine Äquivalenzrelation auf NFAs. Außerdem gilt, dass zwei NFAs isomorph sind, genau dann, wenn sie, bis auf die Namen der Zustände, übereinstimmen.

Satz 6.11 (Isomorphiesatz für DFAs)**.** Es sei $L \subseteq \Sigma^*$ eine reguläre Sprache und k der Index der Kongruenz $\equiv_L$. Jeder DFA A, der L akzeptiert und k Zustände hat, ist isomorph zu A_L.

Beweis. Wir konstruieren einen Isomorphismus vom Äquivalenzklassen-Automaten nach A. Es sei $A_L = (Q_L, q_{0L}, \to_L, Q_{FL})$ der Äquivalenzklassen-Automat mit Zuständen $Q_L = \{[u_1]_{\equiv_L}, \ldots, [u_k]_{\equiv_L}\}$ und $A = (Q, q_0, \to, Q_F)$ mit $|Q| = k$ und $L(A_L) = L(A)$. Definiere $\beta : Q_L \to Q$ so, dass $\beta([u_i]_{\equiv_L})$ der Zustand $q \in Q$ ist, mit $q_0 \xrightarrow{u_i} q$. Dann ist β eine wohl-definierte Abbildung und man kann zeigen, dass β ein Isomorphismus ist (Übung). □

Damit ist nun gezeigt, dass A_L der eindeutige minimale DFA zur Sprache L ist: Es gibt keinen DFA mit echt weniger Zuständen, der L akzeptiert, und jeder DFA mit gleich vielen Zuständen der die Sprache akzeptiert ist isomorph zu A_L.

Bemerkung. Nicht-deterministische Automaten sind kompakter als deterministische. Es kann vorkommen, dass die Zustandszahl eines NFA deutlich unter dem Index der Sprache liegt.
Betrachte die Sprache $L_{nondet}(n)$. Dies ist die Sprache aller Wörter über $\{0, 1\}$, die 1 an der n-ten Stelle von rechts haben. Einen NFA für die Sprache kann man schnell konstruieren:



Es ist also möglich, einen NFA für $L_{nondet}(n)$ anzugeben, der $n+1$ Zustände hat. Es gibt jedoch *keinen* DFA für $L_{nondet}(n)$, der weniger als 2^n Zustände hat, da der Index von $L_{nondet}(n)$ genau 2^n ist.

Intuitiv kann der NFA *raten*, wann genau die n-te Stelle von rechts erreicht ist und die 1 lesen. Dies kann ein DFA nicht, er kann also nicht entscheiden, ob die gerade gelesene 1 die *richtige* 1 ist. Er muss daher n Bits speichern, wofür er 2^n Zustände braucht.

6.3. Konstruktion des deterministischen minimalen Automaten

Bisher haben wir kein konstruktives Verfahren, um den minimalen DFA für eine gegebene Sprache zu berechnen. Wir wissen zwar, dass der Äquivalenzklassenautomat A_L der minimale DFA ist, um ihn zu konstruieren müssten wir allerdings die Äquivalenzklassen von L kennen.

Wir entwickeln nun ein Verfahren, dass mit einem beliebigen DFA für L startet und den minimalen Automaten für L zurückliefert. Wir lösen also das folgende Berechnungsproblem.

Minimierung	
Gegeben:	Ein DFA A.
Berechne:	Den minimalen DFA A' mit $L(A) = L(A')$.

Unser Algorithmus besteht aus zwei Schritten:

1. Lösche unerreichbare Zustände.
2. Fasse äquivalente Zustände zusammen.

1. Lösche unerreichbare Zustände

Definition 6.12 (Erreichbarkeit in Automaten)**.** Sei $A = (Q, q_0, \to, Q_F)$ ein NFA. Ein Zustand $q \in Q$ ist *erreichbar*, wenn es ein Wort $w \in \Sigma^*$ gibt mit $q_0 \xrightarrow{w} q$.

Bemerkung. Wenn q ein erreichbarer Zustand ist, dann gibt es ein Wort $w \in \Sigma^*$ mit $|w| \leq |Q|$ und $q_0 \xrightarrow{w} q$. Dies ermöglicht es uns, alle erreichbaren Zustände mittels eines Fixpunkt-Algorithmus zu finden (siehe Algorithmus für EMPTYREG).

2. Fasse äquivalente Zustände zusammen

Definition 6.13 (Äquivalente Zustände)**.** Sei $A = (Q, q_0, \to, Q_F)$ ein DFA. Wir nennen Zustände $q_1, q_2 \in Q$ *äquivalent*, $q_1 \sim q_2$, falls für alle $w \in \Sigma^*$ gilt:

$$q_1 \xrightarrow{w} q_f \in Q_F \Leftrightarrow q_2 \xrightarrow{w} q'_f \in Q_F.$$

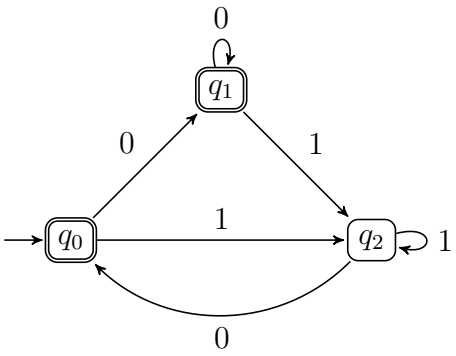
Diese Definition von Äquivalenz ist eine Variante der Nerode-Rechtskongruenz $\equiv_L$, die auf Zuständen statt auf Wörtern arbeitet, wie das folgende Lemma zeigt.

Lemma 6.14. Sei $L = L(A)$ die Sprache eines DFA $A = (Q, q_0, \to, Q_F)$ und u, v Wörter aus Σ^* mit $q_0 \xrightarrow{u} q_1, q_0 \xrightarrow{v} q_2$. Es gilt: $u \equiv_L v \Leftrightarrow q_1 \sim q_2$.

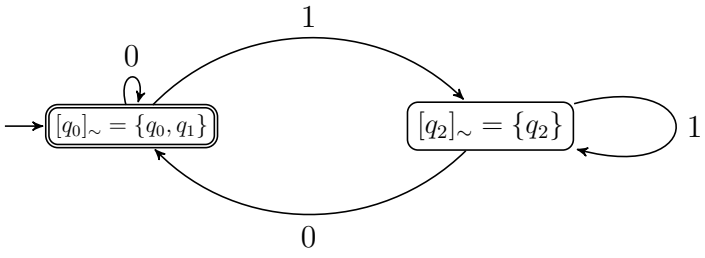
Beweis. Übungsaufgabe. □

Das Lemma zeigt uns an, welche Zustände in einem DFA zusammengelegt werden können: genau die, die äquivalent bezüglich $\sim$ sind. Wir können also aus einem DFA $A = (Q, q_0, \to, Q_F)$ einen potentiell kleineren DFA A' konstruieren, der als Zustände genau die $\sim$-Äquivalenzklassen hat. Der Startzustand ist dann $[q_0]_\sim$ und die Finalzustände sind alle Klassen $[q_f]_\sim$ mit $q_f \in Q_F$. Die Transitionen sind gegeben durch: $[q]_\sim \xrightarrow{a} [q']_\sim$, falls $q \xrightarrow{a} q'$ eine Transition in A ist.

Beispiel 6.15. Sei Σ das Alphabet $\{0, 1\}$. Gegeben sei folgender deterministischer Automat:



Es gilt: $q_0 \sim q_1$ und $q_0 \not\sim q_2$. Für die zweite Beobachtung muss man benutzen, dass $q_0 \xrightarrow{\varepsilon} q_0 \in Q_F$ und $q_2 \xrightarrow{\varepsilon} q_2 \notin Q_F$. Etwas allgemeiner gilt: wenn $q_f \in Q_F \Rightarrow [q_f]_\sim \subseteq Q_F$. Wenn wir die oben beschriebene Konstruktion nun anwenden, erhalten wir:



Bestimmen der äquivalenten Zustände

Wie bestimmen wir algorithmisch, welche Zustandspaare äquivalent sind?

Gemäß Definition gilt $q_1 \sim q_2$, falls für alle $w \in \Sigma^*$

$$q_1 \xrightarrow{w} q_f \in Q_F \Leftrightarrow q_2 \xrightarrow{w} q'_f \in Q_F$$

gilt. Wir schreiben dies um zu

$$\forall i \in \mathbb{N}\ \forall w \in \Sigma^* \text{ mit } |w| = i\colon\ q_1 \xrightarrow{w} q_f \in Q_F \Leftrightarrow q_2 \xrightarrow{w} q'_f \in Q_F\ .$$

Dies liefert uns ein Fixpunktalgorithmus:

- Nehme initial an, dass alle Zustände äquivalent sind, wähle also $R = Q \times Q$.
- Für alle $i = 0, 1, \ldots$:
  - Für alle w mit $|w| = i$:
    - Für alle $(q_1, q_2) \in R$:

      Wenn $q_1 \xrightarrow{w} q'_1 \in Q_F$, aber $q_2 \xrightarrow{w} q'_2 \notin Q_F$ oder anders herum, entferne (q_1, q_2) aus R.

Nach dem „Durchlauf“ des Algorithmus bilden die Paare in R genau die äquivalenten Zustandspaare: wenn $(q, q') \in R$, dann gilt $q \sim q'$.

Optimierungen

- Im Schleifendurchlauf für $i = 0$ wird das Wort ε betrachtet. In diesem Fall werden alle Paare (q_1, q_2) aus R entfernt mit $q_1 \in Q_F$ und $q_2 \notin Q_F$ bzw. anders herum.

  Man kann also direkt mit $R = (Q_F \times Q_F) \cup ((Q \setminus Q_F) \times (Q \setminus Q_F))$ starten.

- Wenn im i-ten Schleifendurchlauf kein weiteres Paar aus R entfernt wird, ist der Fixpunkt erreicht, d.h. auch im $(i + 1)$-ten Schritt würde nichts mehr entfernt werden. Das aktuelle R kann zurückgegeben werden.

  Dies ist zudem spätestens im Durchlauf für $i = |Q|$ der Fall.

- Statt ganze Wörter zu Wörter w zu betrachten, reicht es einzelne Buchstaben a zu betrachten. Genauer gesagt muss man das Kriterium in der Schleife wie folgt anpassen:

  Wenn $q_1 \xrightarrow{a} q'_1$, $q_2 \xrightarrow{a} q'_2$ und $(q'_1, q'_2) \notin R$, dann entferne (q_1, q_2) aus R.

- Da $\sim$ reflexiv und symmetrisch ist, genügt es, Zustandspaare (q_i, q_j) mit $i < j$ zu betrachten.

Algorithm 1 Bestimmung von Äquivalenzklassen (optimiert)

Eingabe: DFA $A = (Q, q_0, \to, Q_F)$
Ausgabe: Äquivalenzrelation $\sim$ auf den Zuständen Q.

```
1: R := (Q_F × Q_F) ∪ ((Q \ Q_F) × (Q \ Q_F))
2: do
3:     R_new := R
4:     for all (q_i, q_j)_{i<j} ∈ R do
5:         for all a ∈ Σ do
6:             if q_i -a-> q_i' ∧ q_j -a-> q_j' ∧ (q_i', q_j') ∉ R then
7:                 R_new := R_new \ {(q_i, q_j), (q_j, q_i)}
8:             fi
9:         od
10:    od
11: while R ≠ R_new
12: return R
```

Aus dem optimierten Algorithmus erhalten wir den folgenden *Table-Filling Algorithmus*, der für händische Berechnungen geeignet ist.

Für den Algorithmus speichern wir eine Tabelle von Zustandspaaren (q_i, q_j) mit $i > j$. Wir erhalten eine trianguläre Tabelle der Größe $\frac{1}{2} \cdot |Q| \cdot (|Q| - 1)$, Der Algorithmus markiert iterativ Paare in der Tabelle, die nicht äquivalent sind. Am Ende entsprechen die unmarkierten Zellen den äquivalenten Zustandspaaren.

Pseudo-Code:

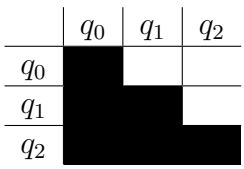
1. Initial: Alle Zustandspaare (Zellen der Tabelle) unmarkiert.
2. Markiere alle Paare $\{q, q'\}$ mit $q \in Q_F$ und $q' \notin Q_F$.
3. Solange es ein unmarkiertes Paar $\{q_1, q_2\}$ und ein $a \in \Sigma$ gibt mit $q_1 \xrightarrow{a} q_1', q_2 \xrightarrow{a} q_2'$, so dass $\{q_1', q_2'\}$ markiert ist:

   markiere $\{q_1, q_2\}$.

Hierbei bezeichnet $\{q_i, q_j\}$ das Zustandspaar (q_i, q_j), falls $i < j$, und (q_j, q_i) andernfalls.
Beachte, dass Zustandspaare (q_i, q_i) immer unmarkiert sind und nicht betrachtet werden müssen.

Beispiel 6.16. Betrachte den DFA aus Beispiel 6.15.

1. Wir konstruieren die Tabelle:



2. Da q_2 im Gegensatz zu q_0 und q_1 kein Endzustand ist, markieren wir (q_0, q_2) und (q_1, q_2).

	q_0	q_1	q_2
q_0	■		×
q_1	■	■	×
q_2	■	■	■

3. Das einzige unmarkierte Zustandspaar ist (q_0, q_1). Betrachten wir die Bedingung aus dem Algorithmus für alle $a \in \Sigma$:

   - $a = 0$: $q_0 \xrightarrow{0} q_1$, $q_1 \xrightarrow{0} q_1$ und (q_1, q_1) ist unmarkiert. Wir müssen also (q_0, q_1) zunächst nicht markieren.
   - $a = 1$: $q_0 \xrightarrow{1} q_2$, $q_1 \xrightarrow{1} q_2$ und (q_2, q_2) ist unmarkiert. Wir müssen also (q_0, q_1) nicht markieren.

4. Die Tabelle aus 2. ist also die finale Tabelle. Sie beschreibt, dass $q_0 \sim q_1$ und $q_0 \not\sim q_2$ gelten.

7. Pumping-Lemma & ultimative Periodizität

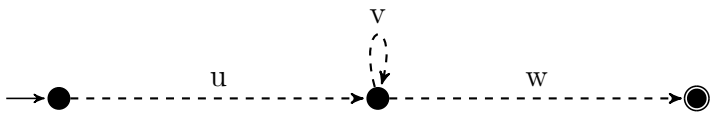
Ziel: Finden einer notwendigen Bedingung für Regularität

- die einfach anzuwenden ist und
- zu anderen Sprachklassen verallgemeinert werden kann.

Wie kann man zum Beispiel leicht beweisen, dass $L_1 = \{a^n b^n \mid n \in \mathbb{N}\}$ nicht regulär ist?

7.1. Das Pumping-Lemma

Idee: Wir zeigen, dass reguläre Sprachen eine strukturelle Eigenschaft haben, welche von L_1 verletzt wird. Die Eigenschaft gibt an, dass Wörter aufgepumpt werden können. Dass heißt wir können Wortteile wiederholen ohne die Sprache zu verlassen.



$$u.v.w \in L \to u.v^i.w \in L, \ \forall i \in \mathbb{N}$$

.

Satz 7.1 (Pumping-Lemma)**.** Für jede reguläre Sprache L gibt es eine *Pumping-Konstante* $p_L \in \mathbb{N}$, so dass $\forall x \in L : |x| \geq p_L$ gilt: Es gibt eine Zerlegung $x = u.v.w$ mit

1. $|v| \geq 1$
2. $|u.v| \leq p_L$
3. $u.v^i.w \in L$ für alle $i \in \mathbb{N}$

Beweis. Sei $L \subseteq \Sigma^*$ regulär mit $L = L(A)$ und $A = (Q, q_0, \to, Q_F)$ ein NFA.
Definiere $p_L := |Q|$.
Betrachte ein Wort $x \in L$ mit $|x| \geq p_L$ und $x = a_1 ... a_d$.
Da x von A akzeptiert wird, gibt es einen Lauf

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots q_j \dots q_k \dots q_{d-1} \xrightarrow{a_d} q_d \in Q_F.$$

Dieser enthält $d+1 > |x| \geq p_L = |Q|$ Zustände.
$\Rightarrow$ Mindestens ein Zustand wiederholt sich: $q_j = q_k$ mit $j < k$.
Wir betrachten die erste der Wiederholungen, das heißt $q_0, ..., q_{k-1}$ sind alle verschieden. Definiere:

- $u := a_1...a_j$.
- $v := a_{j+1}...a_k$.
- $w := a_{k+1}...a_d$.

Damit gilt:

1. $v \neq \epsilon$ da $j < k$.
2. $|u.v| \leq p_L$ da $q_0 \dots q_{k-1}$ alle verschieden sind.
3. Der Automat A kann die $q_j = q_k$ - Schleife beliebig oft wiederholen und akzeptieren.

□

Das Pumping-Lemma wird normalerweise in Kontraposition verwendet, um zu zeigen dass eine Sprache L nicht regulär ist.

- Angenommen L wäre regulär. Dann erfüllt L das Pumping-Lemma und es gibt eine Pumping-Konstante p_L.
- Wähle ein Wort $x \in L$ (üblicherweise abhängig von p_L).
- Zeige, dass jede Zerlegung $x = u.v.w$ mindestens eine der Eigenschaften 1. – 3. verletzt:
  - Betrachte eine beliebige Zerlegung $x = u.v.w$ die 1. und 2. erfüllt.
  - Wähle ein i, so dass $u.v^i.w \notin L$, also ist 3. verletzt.
- Nach dem Pumping-Lemma müsste $u.v^i.w \in L$ für alle i gelten. Wir erhalten einen Widerspruch zum Pumping-Lemma — L ist nicht regulär.

Beispiel 7.2. Betrachte $L_1 = \{a^n b^n \mid n \in \mathbb{N}\}$.

Angenommen L_1 sei regulär und erfüllt damit das Pumping-Lemma. Es sei p_{L_1} die zugehörige Pumping-Konstante.

Wir wählen das Wort $x = a^{p_{L_1}} b^{p_{L_1}}$. Es gilt $|x| \geq p_{L_1}$.

Sei $x = u.v.w$ eine beliebige Zerlegung mit $|v| \geq 1$ und $|u.v| \leq p_{L_1}$.

Da $|u.v| \leq p_{L_1}$, wissen wir, dass u und v nur aus a's bestehen.

Wähle $i = 0$: $x' = u.v^0.w = u.w = a^{|u|}.a^{p_{L_1}-|u|-|v|}.b^{p_{L_1}} = a^{p_{L_1}-|v|}.b^{p_{L_1}}$.

Da $|v| \geq 1$ gilt $x' \notin L_1$. Dies ist ein Widerspruch zum Pumping-Lemma. L_1 ist also nicht regulär. □

Beispiel 7.3. Betrachte $L_2 = \{w \in \{a,b\}^* \mid |w|_a = |w|_b)\}$.
Angenommen L_2 wäre regulär, dann wäre $L_2 \cap a^*b^* = \{a^n b^n \mid n \in \mathbb{N}\} = L_1$ auch regulär. Wir haben gerade gezeigt, dass L_1 nicht regulär ist, also kann L_2 nicht regulär sein. □

Beispiel 7.4. Betrachte $L_3 = \{yy \mid y \in \{a,b\}^*\}$.

Angenommen L_3 wäre regulär und erfüllt damit das Pumping-Lemma. Dann existiert eine Pumping-Konstante $p_{L_3} \in \mathbb{N}$ wie im Pumping-Lemma.

Betrachte das Wort $x := a^{p_{L_3}} b . a^{p_{L_3}} b \ \in L_3$.

Da $|x| \geq p_{L_3}$ gilt, gibt es eine Dekomposition $x = u.v.w$ welche 1. - 3. erfüllt. In dieser bestehen u und v nur aus a's (wegen 2.).

Wähle $i = 2$, betrachte also $x' = u.v.v.w = a^{p_{L_3}+|v|} b . a^{p_{L_3}} b$. Gemäß 3. gilt $x' \in L_3$, jedoch ist x' nicht von der Form yy und damit nicht in L_3.

Widerspruch, L_3 ist also nicht regulär! □

Bemerkung. Das Pumping-Lemma ist nur eine notwendige Bedingung für Regularität, keine hinreichende (es ist also kein „genau dann wenn“). Beispielsweise erfüllt die Sprache

$$\left\{ab^j c^j \;\middle|\; j \in \mathbb{N}\right\} \cup \left\{a^i b^j c^k \;\middle|\; i,j,k \in \mathbb{N}, i \neq 1\right\}$$

das Pumping-Lemma, obwohl sie nicht regulär ist.

Bemerkung. Betrachte die folgende Umschreibung des Pumping-Lemmas:
$\forall p_L \in \mathbb{N}\ \exists x \in L$ mit $|x| \geq p_L$
$\forall$ Dekompositionen $x = u.v.w$ mit $|v| \geq 1$ und $|u.v| \leq p_L$
$\exists i \in \mathbb{N} : u.v^i.w \notin L$.
$\Rightarrow L$ ist nicht regulär.

Dies ist ein Spiel zwischen Defender ($\forall$) und Spoiler ($\exists$):

Spoiler:	Ich glaube, L ist nicht regulär.
Defender:	Warum nicht? Hier ist mein p_L.
Spoiler:	Glaube ich nicht, hier ist mein Wort $x \in L$ mit $\|x\| \geq p_L$. Funktioniert Dein p_L damit?
Defender:	Klar, betrachte die Zerlegung $x = u.v.w$ mit $\|v\| \geq 1$ und $\|u.v\| \leq p_L$.
Spoiler:	Ah, aber nun hast Du verloren, denn hier ist $i \in \mathbb{N}$ mit $u.v^i.w \notin L$.

Das Pumping-Lemma besagt:

- Wenn L regulär ist, hat der Defender eine Gewinnstrategie.

  (Eine Gewinnstrategie ist eine Strategie mit der man gewinnt, egal wie der Gegner spielt.)

- In Kontraposition angewandt sagt es:
  Wenn Spoiler eine Gewinnstrategie hat, dann ist L nicht regulär.

7.2. Ultimative Periodizität

Ziel: Formuliere die Idee, dass reguläre Sprachen nicht (bzw. nur modulo einer festen Zahl) zählen können.

Definition 7.5 (Ultimativ periodisch)**.** Eine Menge $U \subseteq \mathbb{N}$ heißt *ultimativ periodisch*, falls:

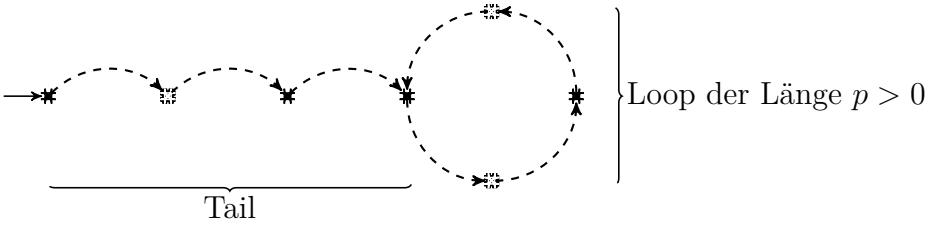
$$\exists n \geq 0 \ \exists p > 0 : \forall m \geq n : \quad m \in U \text{ gdw. } m + p \in U.$$

Idee: Bis auf ein Anfangsstück liegen die Zahlen innerhalb bzw. außerhalb von U nach einem sich wiederholenden Muster.

Satz 7.6. Sei $L \subseteq a^*$. Dann ist L regulär gdw. $\mathit{length}(L) := \{|w| \mid w \in L\}$ ist ultimativ periodisch.

Beweis. "⇒": Sei L regulär, dann gilt $L = L(A)$ für einen DFA A.

Da es nur einen Buchstaben gibt, hat A die Form:



Wähle $n = |\text{Tail}| =$ Zahl der Transitionen in Tail und $p = |\text{Loop}|$.

"⇐": Konstruiere einen DFA mit Tail der Länge n und Loop der Länge p. □

Korollar 7.7. Sei $L \subseteq \Sigma^*$ regulär. Dann ist $\mathit{length}(L)$ ultimativ periodisch.

Beweis. Betrachte den Homomorphismus, der induziert wird durch $h : \Sigma \to \{a\}$ (mit $h(b) = a, \ \forall b \in \Sigma$). Hier ist $h(w) = a^{|w|}$.

Da h die Länge erhält gilt: $\mathit{length}(L) = \mathit{length}(h(L))$. Aber $h(L) \subseteq a^*$ ist eine reguläre Sprache (da reguläre Sprachen abgeschlossen sind unter Homomorphismen). Also ist mit obigem Satz $\mathit{length}(h(L))$ ultimativ periodisch und so auch $\mathit{length}(L)$. □

Bemerkung. Ultimativ periodische Mengen sind nur ein erster Schritt auf dem Weg zur

- Theorie der Parikh-Bilder,
- semi-linearen Mengen,
- und Presburger-Arithmetik.

Im Parikh-Bild eines Worter zählt man die Häufigkeit aller Buchstaben (und vergisst dabei die Reihenfolge), z.B.

$$\text{Parikh}(abaacbcc) = \begin{pmatrix} 3 \\ 2 \\ 3 \end{pmatrix} .$$

Hauptergebnisse:

- Parikh(CFL) = Parikh(REG_Σ) = semi-lineare Menge = Presburger-definierte Mengen.
- Abschluss unter $\cup$, $\cap$, $^-$, *, h, h^{-1}.

Teil III.

Kontextfreie Sprachen

8. Reduktionssysteme, Grammatiken und Chomsky-Hierarchie

Ziel: In diesem Kapitel führen wir Reduktionssysteme und ihr Nutzen in der theoretischen Informatik im Hinblick auf Grammatiken ein. Wir werden sehen, dass verschiedene Einschränkungen auf Reduktionssystemen zu unterschiedlichen Arten von Grammatiken führen, die von der Chomsky-Hierarchie kategorisiert werden.

8.1. Ersetzungssysteme

Sei für den Verlauf dieses Kapitels $\mathbb{A}$ ein beliebiges Alphabet. Die Idee ist, Wörter über $\mathbb{A}$ durch andere Wörter unter Verwendung von bestimmten *Regeln* zu ersetzen.

Definition 8.1. Ein *Ersetzungssystem* über $\mathbb{A}$ ist ein Paar $E = (\mathbb{A}, P)$ mit $P \subseteq \mathbb{A}^\star \times \mathbb{A}^\star$. Elemente von P werden *Produktionen* oder *Ersetzungsregeln* genannt. Wir schreiben $u \to v$ für $(u, v) \in P$.

Betrachte das folgende Reduktionssystem E_1:

$$E_1 = (\underbrace{\{a, \ldots, z\}}_{\mathbb{A}}, \underbrace{\{(kann, muss)\}}_{P})$$

Die Idee ist, Vorkommen des Wortes *kann* durch das Wort *muss* zu ersetzen. Formal führen wir diese Ersetzungen als *Ableitungen* über $\mathbb{A}^\star$ ein.

Definition 8.2. Bezeichne mit $\Rightarrow_E \subset \mathbb{A}^\star \times \mathbb{A}^\star$ die Relation, die *Ableitbarkeit* über E beschreibt. Seien $w, v \in \mathbb{A}^\star$. Dann gilt $w \Rightarrow_E v$ genau dann, wenn es Worte $w_1, w_2 \in \mathbb{A}^\star$ und eine Produktion $u \to z \in P$ gibt, sodass $w = w_1 u w_2$ und $v = w_1 z w_2$.

Wir schreiben auch einfach nur $\Rightarrow$ anstelle $\Rightarrow_E$, wenn das zu Grunde liegende Ersetzungssystem klar ist. Weitere Schreibweisen sind $\to_E$ (bzw. $\to$) in Anlehnung an die Notation für die Produktionsregeln, wenn eine klare Unterscheidung nicht erforderlich ist oder $\vdash_E$ (bzw. $\vdash$).

Mit $\Rightarrow^*$ ist wie üblich die reflexiv-transitive Hülle von $\Rightarrow$ gemeint. Es gilt also $w \Rightarrow^* v$ wenn es eine *Ableitung*, eine Folge von Ersetzungsschritten $w = z_0 \Rightarrow \cdots \Rightarrow z_n = v$, gibt. Wir nennen n die Länge der Ableitung.

Wir machen folgende Beobachtungen:

i) Wenn $w = v$, dann gibt immer eine Ableitung der Länge 0 und es gilt $w \Rightarrow^* w = v$.

ii) Für $w \neq v$ und $w \Rightarrow^* v$ gibt es eine Ableitung der Länge $n > 0$.

iii) Falls $w \Rightarrow v$, leitet w das Wort v *direkt* ab.

iv) Wir schreiben $w \not\Rightarrow$, falls sich keine Ersetzungsregel auf w anwenden lässt, es also kein v gibt, sodass w das Wort v direkt ableitet.

Beispiel 8.3. Betrachte $E_1 = (\{a, \ldots, z\}, \{(kann, muss)\})$ wie oben. Die Ableitungen maximaler Länge ersetzen *alle* Vorkommen von *kann* mit dem Wort *muss*. Im Allgemeinen ist die Reihenfolge der Ersetzungen aber nicht definiert.

1. Ich kann arbeiten $\Rightarrow_{E_1}$ Ich muss arbeiten

2. Ich kann arbeiten und ich kann Sport machen $\Rightarrow_{E_1}$ Ich muss arbeiten und ich kann Sport machen
   oder
   Ich kann arbeiten und ich kann Sport machen $\Rightarrow_{E_1}$ Ich kann arbeiten und ich muss Sport machen

3. Ich kann arbeiten und ich kann Sport machen $\Rightarrow^*_{E_1}$ Ich muss arbeiten und ich muss Sport machen

Beispiel 8.4. Sei $E_2 = (\mathbb{A}, \{(a, aba)\})$. Mit diesem Ersetzungssystem erhalten wir unendliche Ableitungen. Zum Beispiel:

$$cac \Rightarrow_{E_2} cabac \Rightarrow_{E_2} cababac \Rightarrow_{E_2} \cdots$$

Zufälligerweise ist es hier unerheblich, welches a im zweiten Schritt zuerst ersetzt wird. Dies ist im Allgemeinen nicht so. Betrachte ein anderes Wort $w = caac$. Wenn wir immer das erste (das letzte) Vorkommen von a ersetzen, erhalten wir

$$\begin{aligned} caac &\Rightarrow_{E_2} cabaac \Rightarrow_{E_2} cababaac \\ \text{bzw.}\quad caac &\Rightarrow_{E_2} caabac \Rightarrow_{E_2} caababac \end{aligned}$$

Es gibt noch unendlich viele weitere Ableitungen.

Definition 8.5. Die Sprache $L(w, E) := \{v \in \mathbb{A}^* \mid w \Rightarrow^*_E v\}$ ist die Sprache aller aus w mit dem Ersetzungssystem E ableitbaren Worte.

Die Ableitungen von w können als Baum mit Wurzel w dargestellt werden. Die Kindknoten eines Wortes u sind die direkt aus u ableitbaren Worte. Abbildung 8.1 zeigt den (unendlichen) Ableitungsbaum für Beispiel 8.4.

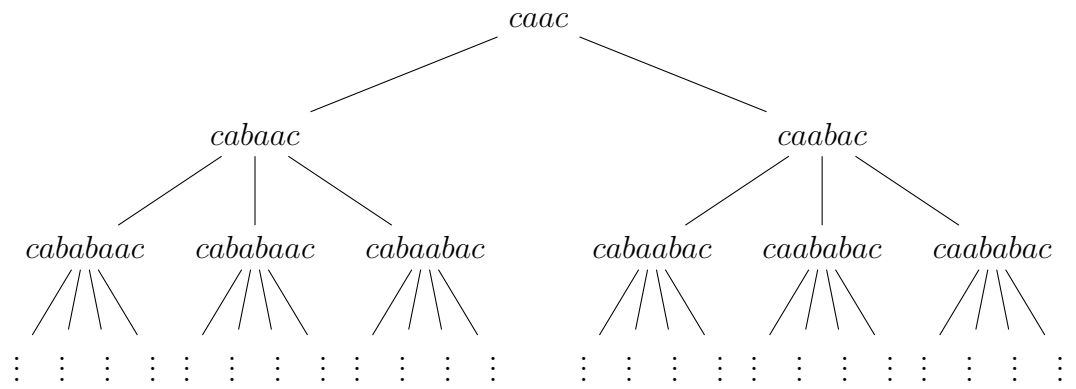


Abbildung 8.1.: Ableitungsbaum für $L(caac, E_2)$

Beispiel 8.6. Betrachte das Ersetzungssystem $E_3 = (\mathbb{A}, \{(ba, ab)\})$ und das Wort $w = baacba \in \mathbb{A}^\star$. Wir beobachten, dass es ein n gibt, für das alle Ableitungen von w mit den Regeln aus E_3 eine Länge von höchstens n haben. Offenbar ist $n = 3$ mit folgender Ableitung:

$$baacba \vdash_{E_3} abacba \vdash_{E_3} abacab \vdash_{E_3} aabcab \ .$$

Die Zahl $(n + 1)$ ist auch die Tiefe des Ableitungsbaumes. Demnach ist die Sprache $L(w, E_3)$ endlich.

Beispiel 8.7. Für $E_4 = (\mathbb{A}, \{(a, b), (a, c), (c, d), (b, e), (d, e)\})$ sehen wir sogar, dass $L(w, E_4)$ *immer* endlich ist, unabhängig von der Wahl von w. Eine Beispielableitung ist

$$a \Rightarrow_{E_4} c \Rightarrow_{E_4} d \Rightarrow_{E_4} e \ .$$

Beispiel 8.8. In Beispiel 8.6 haben wir gesehen, dass eine Sprache $L(w, E)$ endlich ist, wenn der dazugehörige Ableitungsbaum endlich ist. Nun sehen wir, dass die umgekehrte Aussage *nicht* gilt. Sei $E_5 = (\mathbb{A}, \{(x, y), (y, x), (x, a), (y, b)\})$. Abbildung 8.2 zeigt den unendlichen Ableitungsbaum für $L(x, E_5)$ mit der unendlichen Ableitung $x \Rightarrow_{E_5} y \Rightarrow_{E_5} x \Rightarrow_{E_5} y \Rightarrow_{E_5} \cdots$. Allerdings ist $L(x, E_5) = \{x, y, a, b\}$ offensichtlich eine endliche Sprache.

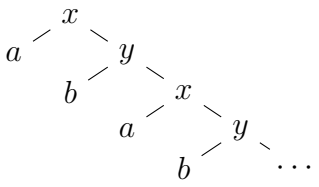


Abbildung 8.2.: Ableitungsbaum für $L(x, E_5)$

8.2. Formale Grammatiken

Formale Grammatiken sind Ersetzungssysteme mit zusätzlicher Struktur. Buchstaben werden in *Nichtterminale* und *Terminale* eingeteilt. Erste können als Hilfssym-

bole für die Ableitung eines Wortes verstanden werden, während letztere das Alphabet für die produzierten Wörter bilden. Per Konvention wird oft das Nichtterminal S als Startsymbol einer Ableitung verstanden. Dementsprechend reden wir über Sprachen $L(S, E_G)$, wobei E_G ein spezielles Ersetzungssystem ist, das nur Wörter ersetzt, in denen Nichtterminale vorkommen.

Formale Grammatiken wurden ursprünglich von Noam Chomsky im Jahr 1959 definiert. Die Unterscheidung zwischen Nichtterminalen und Terminalen ist auch in der Linguistik von Bedeutung. Begriffe wie Subjekt, Prädikat, Objekt können als Nichtterminale aufgefasst werden, während die Buchstaben des deutschen Alphabets (oder der jeweiligen zu betrachtenden Sprache) die Terminale bilden.

$$\text{[Subjekt] [Prädikat] [Objekt]} \Rightarrow^* \text{Captain Picard kommandiert die Enterprise}$$

Definition 8.9. Eine *formale Grammatik* ist ein Viertupel $G = (N, \Sigma, P, S)$ mit

i) einem Alphabet N von *Nichtterminalen*,

ii) einem Alphabet Σ von *Terminalen*, wobei $N \cap \Sigma = \emptyset$ gilt,

iii) einer endlichen Menge von Produktionsregeln $P \subseteq (N \cup \Sigma)^+ \times (N \cup \Sigma)^\star$, mit der Einschränkung, dass die linke Seite einer Regel mindestens ein Nichtterminal enthält $((\alpha, \beta) \in P \Rightarrow \alpha \notin \Sigma^\star)$

iv) und dem Startsymbol $S \in N$.

Wir beobachten, dass $E_G = (N \cup \Sigma, P)$ ein wohldefiniertes Ersetzungssystem (siehe Definition 8.1) ist. Wir benutzen die Notation $\alpha \to \beta$ für Elemente von P und die Relation $\Rightarrow_{E_G}$ für Ableitungen dementsprechend. Produktionen dahingehend einzuschränken, dass sie auf der linken Seite mindestens ein Nichtterminal stehen haben müssen, hat zwei Vorteile: Zum einen verhindert es, Worte aus dem Nichts zu erzeugen. Zum anderen kann ein Wort aus $\Sigma^\star$ nicht mehr geändert werden. Dies motiviert die Definition der Sprache $L(G)$ als die Menge der Worte, die nur aus Terminalen bestehen.

Definition 8.10. Sei $G = (N, \Sigma, P, S)$ eine Grammatik Eine *Satzform* von G ist ein Wort α aus Terminalen und Nichtterminalen über $(N \cup \Sigma)^\star$. Wenn α keine Nichtterminale enthält (d.h. $\alpha \in \Sigma^\star$), nennen wir das Wort *terminal*. Es ist üblich, Satzformen mit griechischen, Terminalwörter mit römischen Buchstaben zu benennen. Die Sprache $L(G)$, die von G produziert wird, ist wie folgt definiert:

$$L(G) = \{w \in L(S, E_G) \mid w \text{ ist terminal}\} \text{ mit } E_G = (N \cup \Sigma, P)$$

Beispiel 8.11. Sei $G_1 = (N_1, \Sigma_1, P_1, S_1)$ eine Grammatik mit

$$\begin{aligned} N_1 &= \{S_1\} \\ \Sigma_1 &= \{a, b\} \\ P_1 &= \{S_1 \to \varepsilon, S_1 \to aS_1b\} \end{aligned}$$

Abbildung 8.3 zeigt den Ableitungsbaum für $L(G_1) = L(S_1, E_{G_1})$. Die linken Ableitungen nutzen die Regel $S_1 \to \varepsilon$ und die rechten Ableitungen nutzen die Regel $S_1 \to aS_1b$. Die terminalen Wörter sind genau die Blätter dieses Baumes $(\varepsilon, ab, aabb, \dots)$. Dementsprechend ist die von G_1 produzierte Sprache $L(G_1) = \{a^n b^n \mid n \in \mathbb{N}\}$.

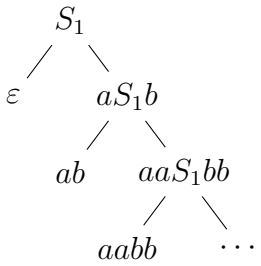


Abbildung 8.3.: Ableitungsbaum für $L(G_1)$

Grammatiken sind sehr nützlich, um gewünschte Strukturen in den Texten zu formalisieren. Die meisten Programmiersprachen sind durch formale Grammatiken definiert, die den Aufbau eines gültigen Programmcodes beschreiben. Das folgende Beispiel zeigt, wie eine einfache Grammatik die korrekte Verwendung von Klammern beschreiben kann.

Beispiel 8.12. Betrachte die Sprache L_{Dyck} mit

$$L_{Dyck} = \{w \in \{(,)\}^\star \mid |w|_(= |w|_) \wedge \forall u \in \mathsf{prefix}(w) : |u|_(\geq |u|_)\}$$

wobei $|w|_a$ die Anzahl der Vorkommen von a in w bezeichnet und $\mathsf{prefix}(w)$ die Menge der Worte x ist, sodass es ein Wort y über dem gleichen Alphabet mit $xy = w$ gibt. Wir können die Grammatik $G_2 = (N_2, \Sigma_2, P_2, S_2)$ wie folgt konstruieren

$$\begin{aligned} N_2 &= \{S_2\} \\ \Sigma_2 &= \{(,)\} \\ P_2 &= \{S_2 \to \varepsilon, S_2 \to (S_2), S_2 \to S_2S_2\} \end{aligned}$$

Es gilt in der Tat $L(G_2) = L_{Dyck}$. (Der Beweis ist eine gute Übung.)

8.3. Die Chomsky-Hierarchie

Bis hier hin dürfen Grammatiken in ihren Produktionsregeln auf der linken Seite beliebige Kombinationen von Terminalen und Nichtterminalen haben, solange es wenigstens ein Nichtterminal gibt. In diesem Abschnitt werden wir weitere Einschränkungen definieren, von denen wir sehen werden, dass sie direkten und starken Einfluss auf die Struktur der Worte haben, die diese Grammatiken erzeugen können. Die Grammatiken sowie die von ihnen erzeugen Sprachen werden dahingehend in Klassen eingeteilt, was uns zur *Chomsky-Hierarchie* führt. Während wir eine Reihe

verschiedener Klassen für Grammatiken finden können, unterscheiden wir typischerweise nur vier große Sprachklassen: Chomsky Typ-0 (uneingeschränkt) bis Typ-3 (am weitesten eingeschränkt). Wir beginnen damit, Einschränkungen für die Produktionsregeln von Grammatiken zu definieren.

Definition 8.13. Eine Produktion $\alpha \to \beta \in P$ einer Grammatik $G = (N, \Sigma, P, S)$ heißt

i) *linkslinear*, wenn $\alpha \in N$ und $\beta \in N\Sigma^\star \cup \Sigma^\star$.
Auf der linken Seite steht genau ein Nichtterminal und auf der rechten Seite befindet sich höchstens ein Nichtterminal, welches ganz links stehen muss.

ii) *rechtslinear*, wenn $\alpha \in N$ und $\beta \in \Sigma^\star N \cup \Sigma^\star$.
Auf der linken Seite steht genau ein Nichtterminal und auf der rechten Seite befindet sich höchstens ein Nichtterminal, welches ganz rechts stehen muss.

iii) *linear*, wenn $\alpha \in N$ und $\beta \in \Sigma^\star N \Sigma^\star \cup \Sigma^\star$.
Auf der linken Seite steht genau ein Nichtterminal und auf der rechten Seite befindet sich höchstens ein Nichtterminal.

iv) *kontextfrei*, wenn $\alpha \in N$.
Auf der linken Seite steht genau ein Nichtterminal und die rechte Seite wird nicht eingeschränkt (insbesondere darf v das leere Wort ε sein).

v) *kontextsensitiv*, wenn $\alpha = \gamma Y \delta$, $\beta = \gamma\eta\delta$ mit $Y \in N, \gamma, \delta, \eta \in (N \cup \Sigma)^\star$ und $\eta \neq \varepsilon$.
Ein Nichtterminal Y auf der linken Seite im Kontext von γ und δ wird durch die nicht-leere Satzform η ersetzt.

vi) *monoton*, wenn $|\alpha| \leq |\beta|$.
Die Produktion verkürzt die Satzform nicht.

Beispiel 8.14. Betrachte $G = (\{S, X\}, \{a, b\}, P, S)$ Die folgende Tabelle zeigt einige Beispiele für Produktionen in P mit ihren jeweiligen Eigenschaften:

Produktion	Eigenschaften
$S \to \varepsilon$	linkslinear, rechtslinear, linear, kontextfrei, nicht kontextsensitiv, nicht monoton
$X \to a$	alle der sechs Eigenschaften i) – vi)
$S \to aX$	rechtslinear, linear, kontextfrei, kontextsensitiv und monoton nicht linkslinear
$S \to Sb$	linkslinear, linear, kontextfrei, kontextsensitiv und monoton nicht rechtslinear
$S \to aSb$	linear, kontextfrei, kontextsensitiv und monoton nicht links- oder rechtslinear
$aSb \to aXaXb$	kontextsensitiv und monoton nicht kontextfrei oder gar linear
$aXb \to bSXa$	monoton und sonst nichts
$aXb \to Xa$	keine der sechs oben genannten Eigenschaften

Wir können zwischen den oben genannten Eigenschaften einige Beziehungen ableiten. Zum Beispiel sind lineare Produktionen auch kontextfrei, was direkt aus der Definition folgt. Kontextfreie Produktionen mit $\beta \neq \varepsilon$ sind auch kontextsensitiv (mit leerem Kontext, also $\gamma = \delta = \varepsilon$) und jede kontextsensitive Produktion ist notwendigerweise monoton. Wir dehnen die Charakterisierung der Produktionen nun auf ganze Grammatiken aus.

Definition 8.15. Eine Grammatik $G = (N, \Sigma, P, S)$ heißt

i) *linkslinear*, wenn *alle* Produktionen linkslinear sind,

ii) *rechtslinear*, wenn *alle* Produktionen rechtslinear sind,

iii) *regulär*, wenn *alle* Produktionen linkslinear oder *alle* Produktionen rechtslinear sind,

iv) *linear*, wenn *alle* Produktionen linear sind,

v) *kontextfrei*, wenn *alle* Produktionen kontextfrei sind,

vi) *kontextsensitiv*, wenn *alle* Produktionen bis auf $S \to \varepsilon$ kontextsensitiv sind (in diesem Fall darf S nicht auf der rechten Seite einer Produktion vorkommen),

vii) *monoton* (oder nichtverkürzend), wenn *alle* Produktionen bis auf $S \to \varepsilon$ monoton sind (in diesem Fall darf S nicht auf der rechten Seite einer Produktion vorkommen).

Für kontextsensitive und monotone Grammatiken brauchen wir die Ausnahme für $S \to \varepsilon$, um die Produktion des leeren Wortes zu erlauben. Ohne diese Ausnahme wäre $\varepsilon \notin L(G)$ für alle kontextsensitiven / monotonen Grammatiken G erzwungen. Dennoch darf S nicht gleichzeitig auch auf der rechten Seite einer Produktion vorkommen, da dies sonst die Monotonieeigenschaft verletzt. Zum Beispiel die Produktion $aXb \to aSb$ in Kombination mit $S \to \varepsilon$ würde die Ableitung $aXb \Rightarrow^* ab$ mit $|aXb| \not\leq |ab|$ erlauben, obwohl alle Produktionsregeln den Einschränkungen genügen.

Nun können wir *Grammatiken* in Kategorien wie (links-/rechts-)linear, kontextfrei, kontextsensitiv oder monoton einteilen. Wir ändern noch einmal die Perspektive und nehmen eine ähnliche Kategorisierung für *Sprachen* vor. Zur Erinnerung: Bei der Übertragung der Eigenschaften auf Grammatiken argumentierten wir über *alle* Produktionen. Beispielsweise ist eine Grammatik kontextfrei, wenn *alle* Produktionen kontextfrei sind. Für Sprachen soll es hingegen ausreichen, dass es eine Grammatik G *gibt*, die die Sprache erzeugt. Es mag dabei viele weitere Grammatiken eines anderen Typs geben, die die gleiche Sprache erzeugen.

Definition 8.16. Sei Σ ein Alphabet. Eine Sprache $L \subseteq \Sigma^\star$ heißt

i) *regulär (Chomsky Typ-3)*, falls es eine linkslineare oder rechtslineare Grammatik G mit $L = L(G)$ gibt,

ii) *linear*, falls es eine lineare Grammatik G mit $L = L(G)$ gibt,

iii) *kontextfrei (Chomsky Typ-2)*, wenn es eine kontextfreie Grammatik G mit $L = L(G)$ gibt,

iv) *kontextsensitiv (Chomsky Typ-1)*, wenn es eine kontextsensitive Grammatik G mit $L = L(G)$ gibt,

v) *nichtverkürzend*, wenn es eine monotone Grammatik G mit $L = L(G)$ gibt,

vi) *Chomsky Typ-0*, wenn es *irgendeine* Grammatik G mit $L = L(G)$ gibt.

Die Klasse der kontextfreien Sprachen wird mit CFL (*context-free languages*) bezeichnet.

Beispiel 8.17. Betrachte $\Sigma = \{a, b\}$ und die Sprache $L_3 = \{a^n b \mid n \in \mathbb{N}\}$. Die Sprache kann von der folgenden rechtslinearen Grammatik erzeugt werden: $G_3 = (N_3, \Sigma, P_3, S)$ mit

$$\begin{aligned} N_3 &= \{S\} \\ P_3 &= \{S \to aS, S \to b\} \end{aligned}$$

Dementsprechend ist L_3 regulär. Die folgende linkslineare Grammatik erzeugt ebenfalls L_3: $G_3' = (N_3', \Sigma, P_3', S)$

$$\begin{aligned} N_3' &= \{A, S\} \\ P_3' &= \{S \to Ab, A \to Aa, A \to \varepsilon\} \end{aligned}$$

In der Tat kann gezeigt werden, dass es für *jede* reguläre Sprache eine rechts- sowie eine linkslineare Grammatik gibt. Andererseits ist die Sprache einer jeden links- oder rechtslinearen Grammatik im Sinne von Teil II. der Vorlesung regulär. Der Begriff der Regularität einer Sprache aus Definition 8.16 stimmt also mit der Definition der regulären Sprachen aus Teil II. überein. Der Beweis dieser Aussagen ist eine gute Übung.

Beispiel 8.18. Wir erinnern uns an die Grammatik G_1 mit $L_1 = L(G_1) = \{a^n b^n \mid n \in \mathbb{N}\}$. Da beide Produktionen $S_1 \to \varepsilon$ und $S \to aS_1b$ kontextfrei sind, ist G_1 eine kontextfreie Grammatik und L_1 damit eine kontextfreie Sprache. Aus Kapitel 7 wissen wir allerdings, dass es für L_1 keine reguläre Grammatik geben kann.

Beispiel 8.19. Sei $\Sigma = \{a, b, c\}$ und $L_4 = \{a^n b^n c^n \mid n \in \mathbb{N}\}$. Diese Sprache ist nichtverkürzend, da wir folgende monotone Grammatik angeben können: $G_4 = (N_4, \Sigma, P_4, S)$

$$\begin{aligned} & N_4 = \{S, R, B\} & \\ P_4: \quad & S \to \varepsilon & S \to R \\ & R \to aRBc & R \to abc \\ & cB \to Bc & bB \to bb \end{aligned}$$

Eine Ableitung ist beispielsweise $S \Rightarrow R \Rightarrow aRBc \Rightarrow aabcBc \Rightarrow aabBcc \Rightarrow aabbcc$. Man kann zeigen, dass L_4 sogar kontextsensitiv ist, aber nicht kontextfrei.

Notation: Oftmals geben wir bei einer Grammatik nur die Regeln an. Die Nichtterminale und Terminale ergeben sich dann aus diesen. Üblicherweise werden Nichtterminale mit römischen Großbuchstaben bezeichnet und das Startsymbol trägt den Namen S.

Wenn eine kontextfreie Grammatik mehrere Produktionsregeln für das selbe Nichtterminal hat, werden diese zusammengefasst. Beispielweise schreibt man statt $X \to b$ und $X \to aZa$ nur $X \to b \mid aZa$.

9. Das Wortproblem für kontextfreie Sprachen & der CYK-Algorithmus

Im Rest der Vorlesung werden wir uns hauptsächlich mit kontextfreien Sprachen befassen. In diesem Kapitel wollen wir einen Algorithmus kennen lernen, der das Wortproblem für diese löst. Hierbei nehmen wir an, dass eine kontextfreie Grammatik gegeben ist, die die Sprache erzeugt.

Wortproblem für kontextfreie Sprachen (WORDCFL)	
Gegeben:	Kontextfreie Grammatik $G = (N, \Sigma, P, S)$, Terminalwort $w \in \Sigma^*$.
Frage:	Gilt $w \in L(G)$?

9.1. Die Chomsky-Normalform

Kontextfreie Grammatiken lassen sich in verschiedene Normalformen überführen. Normalformen zeichnen sich dadurch aus, dass sie eine besonders einfache Struktur haben. Dies ist einerseits von theoretischem Interesse, da Resultate über CFGs dadurch vereinfacht werden, und andererseits von praktischem Interesse, da Algorithmen für CFGs durch die Struktur effizienter werden.

Der Algorithmus für das Wortproblem, den wir später kennen lernen, geht davon aus, dass die Grammatik in einer bestimmten Normalform gegeben ist.

Definition 9.1 (Chomsky-Normalform)**.** Eine kontextfreie Grammatik ist in *Chomsky-Normalform* (CNF), falls alle Produktionen von der Form $X \to YZ$ (für Nichtterminale Y, Z) oder $X \to a$ (für ein Terminal a) sind.

Anhand der Definition sieht man, dass für eine kontextfreie Grammatik G in CNF $\varepsilon \notin L(G)$ gelten muss, da wir Produktionsregeln der Form $X \to \varepsilon$ nicht erlauben. Um dieses Problem zu umgehen, lässt man üblicherweise die Regel $S \to \varepsilon$ für das Startsymbol zu, verbietet dann aber das Vorkommen von S auf der rechten Seite von Produktionsregeln.

Satz 9.2. Für jede CFG G können wir eine CFG G' in CNF konstruieren, sodass $L(G') = L(G) \setminus \{\varepsilon\}$.

Der Beweis besteht aus zwei Teilen.

1. Zuerst entfernen wir alle ε-Produktionen, also Produktionen der Form $X \to \varepsilon$ aus der Grammatik.
2. Dann bringen wir die verbleibenden Regeln in Chomsky-Normalform.

Eliminierung von ε-Produktionen.

Lemma 9.3. Für jede CFG G können wir eine kontextfreie Grammatik G'' konstruieren, die keine ε-Produktionen enthält und $L(G'') = L(G) \setminus \{\varepsilon\}$ erfüllt.

Der folgende Beweis des Lemmas ist *effektiv*, da er nicht nur die Existenz der Grammatik G' beweist, sondern auch ein Verfahren liefert, um sie zu berechnen.

Beweis. Es sei $G = (N, \Sigma, P, S)$. Wir definieren $\hat{G} := (N, \Sigma, \hat{P}, S)$. Dabei ist $\hat{P}$ die kleinste Menge, die P enthält und abgeschlossen ist unter der folgenden Regel:

Falls $X \to \alpha Y \beta$ und $Y \to \varepsilon$ in $\hat{P}$, dann ist auch $X \to \alpha\beta$ in $\hat{P}$,

Beachte, dass $\hat{P}$ nur endlich viele Produktionen beinhaltet. Der Grund ist, dass die Länge der rechten Seiten in $\hat{P}$ beschränkt ist durch die Länge der rechten Seiten in P. Dies lässt nur endlich viele Produktionen zu.

Nun gilt, dass $L(\hat{G}) = L(G)$. Die Inklusion "$\supseteq$"' ist dabei klar, da $P \subseteq \hat{P}$. Die andere Inklusion gilt, da jede neue Regel, die wir zu $\hat{P}$ hinzufügen, durch die Komposition zweier alter Regeln simuliert werden kann.

Wir werden nun zeigen, dass für jedes $w \in \Sigma^+$ eine Ableitung $S \Rightarrow_{\hat{G}}^* w$ minimaler Länge keine ε-Produktionen nutzt. Dazu sei $S \Rightarrow_{\hat{G}}^* w$ eine solche Ableitung. Angenommen, die Produktion $Y \to \varepsilon$ wird in der Ableitung benutzt. Die Ableitung lässt sich dann schreiben als:

$$S \Rightarrow_{\hat{G}}^* \alpha_1.Y.\alpha_2 \Rightarrow_{\hat{G}} \alpha_1\alpha_2 \Rightarrow_{\hat{G}}^* w \ .$$

Da $w \neq \varepsilon$, gilt $\alpha_1.\alpha_2 \neq \varepsilon$. Daher ist Y nicht das initiale S, sondern ist durch eine Ersetzung $X \to \beta_1 Y \beta_2$ eingeführt worden. Daraus ergibt sich:

$$S \Rightarrow_{\hat{G}}^{\mathrm{m}} \gamma_1 X \gamma_2 \Rightarrow_{\hat{G}} \gamma_1\beta_1 Y \beta_2\gamma_2 \Rightarrow_{\hat{G}}^{\mathrm{n}} \alpha_1 Y \alpha_2 \Rightarrow_{\hat{G}} \alpha_1\alpha_2 \Rightarrow_{\hat{G}}^{\mathrm{k}} w \ .$$

Die Menge $\hat{P}$ der Produktionen ist so konstruiert, dass es auch zwangsläufig die Regel $X \to \beta_1\beta_2$ geben muss. Nun können wir folgende Ableitung konstruieren:

$$S \Rightarrow_{\hat{G}}^{\mathrm{m}} \gamma_1 Y \gamma_2 \Rightarrow_{\hat{G}} \gamma_1\beta_1\beta_2\gamma_2 \Rightarrow_{\hat{G}}^{\mathrm{n}} \alpha_1\alpha_2 \Rightarrow_{\hat{G}}^{\mathrm{k}} w \ .$$

Diese Ableitung von w ist jedoch kürzer als die zuvor betrachtete. Dies ist ein Widerspruch zur Minimalität dieser.

Wir brauchen also um Worte $w \neq \varepsilon$ aus $L(G)$ abzuleiten keine ε-Produktionen. Wir definieren P'' als die Teilmenge von $\hat{P}$, die durch das Entfernen aller ε-Produktionen entsteht, und $G'' = (N, \Sigma, P'', S)$. Es gilt G'' mit $L(G'') = L(G) \setminus \{\varepsilon\}$ wie gewünscht. □

Beispiel 9.4. Betrachte die CFG G mit den Regeln $S \to ABC, A \to a, B \to \varepsilon \mid b, C \to BB$. Es gilt $L(G) = \{a, ab, abb, abbb\}$. Wir erhalten, dass $\hat{P}$ aus den Regeln aus P sowie den neuen Regeln $C \to B$, $C \to \varepsilon$, $S \to AB$, $S \to AC$, $S \to A$ besteht.

Durch Entfernen der ε-Produktionen erhalten wir die Grammatik G'' mit den folgenden Regeln:

$$\begin{aligned} S &\to ABC \mid AB \mid AC \mid A \ , \\ A &\to a \ , \\ B &\to b \ , \\ C &\to BB \mid B \ . \end{aligned}$$

Es gilt $L(G'') = L(G)$.

Überführen in Chomsky-Normalform. Nehmen wir nun an, dass G eine Grammatik ist, die keine ε-Produktionen enthält. Es verbleibt, alle Regeln in Chomsky-Normalform zu überführen.

Dies geschieht in drei Schritten

1. Führe neue Nichtterminale ein, die zu den Terminalen ableiten und ersetze die Vorkommen von Terminalsymbolen in den rechten Seiten von Produktionsregeln durch diese.
2. Entferne sogenannte Unit-Produktionen der Form $X \to Y$.
3. Splitte Regeln, deren rechte Seite aus mehr als zwei Symbolen besteht auf, indem neue Nichtterminale eingeführt werden.

Es sei der Leserin/dem Leser überlassen, als Übung die Details dieses Verfahrens auszuarbeiten. Wir demonstrieren seine Wirkungsweise an zwei Beispielen.

Beispiel 9.5. Betrachte die Grammatik mit den Regeln $S \to XY$, $X \to Y \mid a$, $Y \to aYa \mid b$.

1. Wir fügen zunächst neue Nichtterminale A, B mit den Regeln $A \to a$ und $B \to b$ ein und ändern die Regel $Y \to aYa$ zu $Y \to AYA$. Die Regeln $X \to a$ und $Y \to b$ entsprechen bereits Chomsky-Normaform, hier ist also keine Änderung nötig.
2. Wir entfernen die Unit-Regel $X \to Y$, fügen dafür aber entsprechende „Abkürzungen“ ein. Wir erhalten die Regeln

$$\begin{aligned} S &\to XY \mid YY \ , \\ Y &\to AYA \mid b \ . \end{aligned}$$

3. Wir fügen ein neues Nichtterminal Z ein und ersetzen die Regel $Y \to AYA$ durch $Y \to AZ$, $Z \to YA$.

Insgesamt erhalten wir eine Grammatik in Chomsky-Normalform, die durch die folgenden Regeln gegeben ist.

$$\begin{aligned} S &\to XY \mid YY \ , & X &\to a \ , \\ Y &\to AZ \mid b \ , & A &\to a \ , \\ Z &\to YA \ , & B &\to b \ . \end{aligned}$$

Beispiel 9.6. Betrachte die Sprache $L = \{a^n b^n \mid n \in \mathbb{N}\}$. Eine kontextfreie Grammatik für L ist gegeben durch: $S \to aSb \mid \varepsilon$. Nun wollen wir eine CNF für die Grammatik berechnen. Dazu entfernen wir zuerst die ε-Produktionen. Wir erhalten als neue Regelmenge

$$S \to aSb \mid ab \ .$$

Nun fügen wir Nichtterminale A, B hinzu:

$$\begin{aligned} S &\to ASB \mid AB, \\ A &\to a, \\ B &\to b. \end{aligned}$$

Schließlich fügen wir C hinzu und ersetzen $S \to ASB$ durch $S \to AC$ und $C \to SB$. Insgesamt ergibt sich:

$$\begin{aligned} S &\to AC \mid AB, \\ C &\to SB, \\ A &\to a, \\ B &\to b. \end{aligned}$$

Nützliche Nichtterminale. Analog zu unerreichbaren Zuständen in endlichen Automaten kann es in einer Grammatik Nichtterminale geben, die nichts zur Sprache beitragen und entfernt werden können. Dies wollen wir im folgenden kurz diskutieren.

Definition 9.7 (Nützliche Nicht-Terminale)**.** Sei $G = (N, \Sigma, P, S)$ eine CFG. Ein Nichtterminal $X \in N$ heißt *nützlich*, falls es ein $w \in \Sigma^*$ gibt, mit: $S \Rightarrow^* \alpha X \beta \Rightarrow^* w$, für Satzformen $\alpha, \beta \in (N \cup \Sigma)^*$. Ist dies für ein Nichtterminal nicht erfüllt, heißt dieses *unnütz*.

Lemma 9.8. Für jede CFG G können wir eine kontextfreie Grammatik G' mit $L(G) = L(G')$konstruieren, die keine unnützen Nichtterminale enthält.

Beweisskizze.

- Mit einer Fixpunktberechnung rückwärts bestimmen wir die Nichtterminale, aus denen sich ein Terminalwort ableiten lässt. Alle anderen Nichtterminale, ihre Regeln und die Regeln, auf deren rechten Seiten sie vorkommen, verwerfen wir.
- Mit einer Fixpunktberechnung vorwärts bestimmen wir nun alle Nichtterminale, die in von S ausgehenden Ableitungen auftreten können. Alle anderen Nichtterminale und ihre Regeln verwerfen wir.

□

9.2. Dynamic Programming und der CYK-Algorithmus

Ziel: Zeige dass das Wortproblem in Polynomialzeit gelöst werden kann. Hierzu führen wir die algorithmische Technik der dynamischen Programmierung ein.

Dynamic Programming

- Akkumuliere Informationen über Teilprobleme, um größere Probleme zu lösen.
- Speichere Lösungen für Teilprobleme, um erneute Berechnung zu vermeiden.

Beispiel 9.9 (Fibonacci)**.** *Naiver Algorithmus berechnet Werte mehrfach:*

$$\begin{aligned} fib(5) &= fib(4) + fib(3) \\ &= fib(3) + fib(2) + fib(2) + fib(1) \\ &= \underline{fib(2)} + fib(1) + \underline{fib(2)} + \underline{fib(2)} + fib(1) \ . \end{aligned}$$

Dynamic Programming: Speichere $mem(0) := 0$ und $mem(1) := 1$ und setze $mem(n) := mem(n-1) + mem(n-2)$ für $n = 2, 3, \ldots$ bis zum gewünschten Wert.

Dynamische Programmierung kann als Fixpunktberechnung auf gespeicherten Hilfsinformationen gesehen werden.

Wir betrachten nun erneut das Wortproblem, wobei wir davon ausgehen, dass die Grammatik in Chomsky-Normalform gegeben ist.

Wortproblem für kontextfreie Sprachen (WORDCFL)	
Gegeben:	CFG $G = (N, \Sigma, P, S)$ in CNF, Terminalwort $w \in \Sigma^*$.
Frage:	Gilt $w \in L(G)$?

Idee: Die Teilprobleme bestimmen für jeden Infix v von w und jedes Nichtterminal X, ob $X \Rightarrow^* v$ gilt.

Tabelle: Der Algorithmus trägt die Lösungen in eine $n \times n$ Tabelle ein, $n := |w|$.

Für $i \leq j$: $table(i, j) :=$ Nichtterminale, die $w_i ... w_j$ erzeugen.

Für $i > j$: Keine Einträge.

Der Algorithmus füllt die Tabelleneinträge für alle Infixe von w aus, der Länge nach sortiert, also zunächst für Infixe der Länge 1, dann für Infixe der Länge 2 usw.

Kernidee: Nutze Einträge für kürzere Infixe, um Einträge für längere Infixe zu bestimmen.

Akzeptanz: Der Algorithmus gibt „ja“ zurück, falls $S \in table(1, n)$, weil dann $S \Rightarrow^* w_1 \dots w_n = w$.

Details zum Ausfüllen:

- Initial füllen wir die Tabelle für Infixe der Länge 1 aus. Für jedes i beinhaltet $table(i, i)$ alle Nichtterminale X mit Regel $X \to w_i$.
- Angenommen, wir haben schon bestimmt, welche Nichtterminale die Infixe der Länge $\leq k$ erzeugen.
- Um zu bestimmen, ob X das Infix v der Länge $k + 1$ erzeugt, spalte $v = a_1...a_{k+1}$ in zwei nicht leere Teile $v_1 = a_1...a_n$ und $v_2 = a_{n+1}...a_{k+1}$. Es gibt k Möglichkeiten v zu teilen.

  Wir betrachten alle Regeln $X \to YZ$ und prüfen, ob Y v_1 generiert und Z v_2 generiert. Falls ja, füge X dem Eintrag für v hinzu.

Beispiel 9.10. Sei $G = (\{S, A, B, C\}, \{a, b\}, P, S)$ mit P gegeben durch

$$S \to AB \mid BC\ , \qquad A \to BA \mid a\ ,$$
$$B \to CC \mid b\ , \qquad C \to AB \mid a$$

und $w = baaba$.

Tabelle:

	1	2	3	4	5
1	$\{B\}$	$\{S, A\}$	$\emptyset$	$\emptyset$	$\{A, C, S\}$
2		$\{A, C\}$	$\{B\}$	$\{B\}$	$\{A, C, S\}$
3			$\{A, C\}$	$\{S, C\}$	$\{B\}$
4				$\{B\}$	$\{A, S\}$
5					$\{A, C\}$

Es gilt $w \in L(G)$, da $S \in \{A, C, S\} = table(1, 5)$.

Die formale Beschreibung des Verfahrens befindet sich in Pseudocode 1.

Komplexität: Der Algorithmus besteht aus 3 geschachtelten Schleifen: Die erste Schleife ist für die Länge des Infixes, die zweite Schleife für den Start des Teilwortes und die dritte Schleife für die Aufspaltposition. In Rumpf der innersten Schleife müssen wir zudem über alle Produktionsregeln der Grammatik iterieren. Daraus ergibt sich eine Laufzeit von $O(|w|^3 \cdot |G|)$.

Satz 9.11. WORDCFL lässt sich in $O(|w|^3 \cdot |G|)$ lösen.

Pseudocode 2 CYK-Algorithmus (Cocke, Younger, Kasami '60)

Eingabe: Grammatik $G = (N, \Sigma, P, S)$, Wort $w = a_1...a_n$.
Ausgabe: true gdw. $w \in L(G)$.
Algorithmus:

1: **for all** $i = 1, ..., n$ **do**
2: $\quad table(i, i) := \{X \in N \mid X \to a_i \in P\}$
3: **end for all**
4: **for all** $k = 2, ..., n$ **do**
5: $\quad$ **for all** $i = 1, ..., (n - k) + 1$ **do**
6: $\quad\quad table(i, (i + k) - 1) := \emptyset$
7: $\quad\quad$ **for all** $m = 1, ..., k - 1$ **do**
8: $\quad\quad\quad table(i, (i + k) - 1) := table(i, (i + k) - 1) \cup \{ X \in N \mid X \to YZ$
$\quad\quad\quad\quad$ mit $Y \in table(i, (i+m)-1)$ und $Z \in table((i+m), (i+k)-1) \}$
9: $\quad\quad$ **end for all**
10: $\quad$ **end for all**
11: **end for all**
12: **return true**, wenn $S \in table(1, n)$, sonst **false**

Bemerkung. Wenn wir annehmen, dass die Grammatik konstant (also nicht Teil der Eingabe) ist, so erhalten wir eine Laufzeit von $O(|w|^3)$. Dies ist relevant für das Parsen von Programmen: Die Grammatik, die eine Programmiersprache definiert, ist fest, es werden jedoch für viele verschiedene Programme w Anfragen gestellt („Ist w ein syntaktisch korrektes Programm?“). In der Praxis ist die kubische Laufzeit des CYK-Algorithmus zu langsam. Daher beschränkt man sich oftmals auf Grammatiken einer bestimmten Form, für die das Wortproblem effizienter (nämlich in Linearzeit) gelöst werden kann.

10. Die Greibach-Normalform

Für einen Beweis im nächsten Kapitel der Vorlesung werden wir benötigen, dass sich jede Grammatik in *Greibach-Normalform* überführen lässt. Dies wollen wir in diesem Kapitel beweisen.

Definition 10.1 (Greibach-Normalform)**.** Eine kontextfreie Grammatik ist in *Greibach-Normalform* (GNF), falls alle Produktionen von der Form $A \to a.B_1 \dots B_k$ mit $k \geq 0$ sind.

Satz 10.2. Zu einer CFG G können wir eine kontextfreie Grammatik G' in GNF konstruieren, sodass $L(G') = L(G) \setminus \{\varepsilon\}$.

Beweis. Unter Verwendung von Satz 9.2 können wir annehmen, dass $G = (N, \Sigma, P, S)$ in Chomsky-Normalform ist. Sei $A \in N$ und $a \in \Sigma$. Wir definieren die Sprache:

$$R_{A,a} := \{\beta \in N^* \mid A \Rightarrow^*_{\mathrm{SL}} a.\beta\}.$$

Die starke Linksableitung $\Rightarrow_{\mathrm{SL}}$ ist eine Teilmenge von $(N \cup \Sigma)^* \times (N \cup \Sigma)^*$, definiert durch: $\alpha_1 \Rightarrow_{\mathrm{SL}} \alpha_2$ falls α_2 aus α_1 durch Ersetzung des am weitesten links stehenden Symbols hervorgeht. Falls dieses kein Nichtterminal ist, so sind keine starken Linksableitungen möglich.

Die Sprache $R_{A,a}$ ist eine reguläre Sprache über dem Alphabet N: Die Grammatik

$$(\{B' \mid B \in N\}, N, (\{B' \to C'D \mid B \to CD \in P\} \cup \{B' \to \varepsilon \mid B \to a\}), A')$$

ist links-linear und akzeptiert $R_{A,a}$. Die Grammatik hat für jedes Nichtterminal B aus G ein neues Nichtterminal B'. Die früheren Nichtterminale werden zu Terminalsymbolen. Die Produktionen sind so konstruiert, dass sie immer nur das Nichtterminal ersetzen, welches ganz links steht. Dadurch wird die starke Linksableitung simuliert. Der Ersetzungsprozess kann gestoppt werden, sobald a gefunden ist. [1]

Da $R_{A,a}$ regulär ist, gibt es zu der Grammatik für $R_{A,a}$ auch eine rechts-lineare Grammatik $G_{A,a}$ mit Produktionen: $X \to BY$ oder $X \to \varepsilon$. Wir können annehmen, dass die Nichtterminale aller Grammatiken $G_{A,a}$ paarweise disjunkt sind. Nun konstruieren wir eine Grammatik G_1 aus G, indem wir alle Nicht-Terminale und Produktionen der $G_{A,a}$ hinzufügen. Das Startsymbol bleibt S. Die Produktionen in

[1] Es ist ein *konzeptueller* Schritt, Worte aus Nichtterminale als reguläre Sprachen aufzufassen.

G_1 sind dann von der Form: $X \to \epsilon$, $X \to BY$, $X \to b$.

Es gilt $L(G_1) = L(G)$, da die Regeln der Grammatiken $G_{A,a}$ von S aus nicht erreichbar sind. Die $G_{A,a}$ nutzen neue Nichtterminale und Startsymbole. Sei $T_{A,a}$ das Startsymbol von $G_{A,a}$. Wir konstruieren aus G_1 eine Grammatik G_2, indem wir jede Regel $X \to BY$ durch die Regeln $X \to aT_{B,a}Y$ für alle $a \in \Sigma$ ersetzen. Die Idee hinter dieser Ersetzung: wir raten das Terminalsymbol a, das in einer Ableitung von B aus erzeugt wird. Die möglichen folgenden Nichtterminale, die von B erzeugt werden, werden durch $T_{B,a}$ und die Produktionen aus $G_{B,a}$ erzeugt.

Die Produktionen in G_2 haben nun die Form: $X \to \epsilon$, $X \to b$ oder $X \to aT_{B,a}Y$. Nun eliminieren wir in G_2 alle ε-Produktionen und erhalten G_3. Diese Grammatik ist in GNF und es gilt: $L(G_3) = L(G) \setminus \{\varepsilon\}$. □

Der Beweis von Satz 10.2 liefert ein Verfahren, um eine gegebene Grammatik in Greibach-Normalform umzuwandeln.

Beispiel 10.3. Betrachte die folgende Grammatik G in CNF:

$$\begin{aligned} &S \to AB \mid AC \mid SS, \\ &A \to [, \\ &B \to], \\ &C \to SB. \end{aligned}$$

Wir wollen eine GNF berechnen. Im ersten Schritt, leiten wir dazu die regulären Sprachen $R_{A,a}$ her:

(1) $R_{S,[} = (B \cup C).S^*$

(2) $R_{C,[} = (B \cup C).S^*.B$

(3) $R_{A,[} = \{\varepsilon\} = R_{B,]}$

Alle anderen Sprachen sind leer.

Im zweiten Schritt berechnen wir rechts-lineare Grammatiken für die regulären Sprachen:

(1) $T_{S,[} \to BX \mid CX$, $X \to SX \mid \varepsilon$.

(2) $T_{C,[} \to BY \mid CY$, $Y \to SY \mid BZ$,
$Z \to \varepsilon$.

(3) $T_{A,[} \to \varepsilon$,
$T_{B,]} \to \varepsilon$.

Diese Grammatiken kombinieren wir nun mit G. Danach wenden wir die Ersetzungsregel an.

$$\begin{aligned}
S &\to [T_{A,[}B \mid [T_{A,[}C \mid [T_{S,[}S,\\
T_{S,[} &\to]T_{B,]}X \mid [T_{C,[}X,\\
T_{C,[} &\to]T_{B,]}Y \mid [T_{C,[}Y,\\
T_{A,[} &\to \epsilon,\\
T_{B,]} &\to \epsilon,\\
C &\to [T_{S,[}B,\\
X &\to [T_{S,[}X \mid \epsilon,\\
Y &\to [T_{S,[}Y \mid]T_{B,]}Z,\\
A &\to [,\\
B &\to],\\
Z &\to \epsilon.
\end{aligned}$$

Zum Schluss entfernt man alle ε-Produktionen und erhält die GNF:

$$\begin{aligned}
S &\to [B \mid [C \mid [T_{S,[}S,\\
T_{S,[} &\to] \mid]X \mid [T_{C,[} \mid [T_{C,[}X,\\
T_{C,[} &\to]Y \mid [T_{C,[}Y,\\
C &\to [T_{S,[}B,\\
X &\to [T_{S,[}X \mid [T_{S,[},\\
Y &\to [T_{S,[}Y \mid],\\
A &\to [,\\
B &\to].
\end{aligned}$$

Bemerkung. Wenn wir im Beweis von Satz 10.2 annehmen, dass G keine unnützen Nichtterminale beinhaltet, im Beweis nur die Fälle $R_{A,a} \neq \emptyset$ beachten und darauf achten, dass alle $G_{A,a}$ keine unnützen Nichtterminale beinhalten, dann hat auch G_3 keine unnützen Nichtterminale.

11. Pushdown-Automaten

Ziel: Entwickle automatentheoretisches Verständnis kontextfreier Sprachen.

Motivation: Macht das Beweisen von Abschlusseigenschaften einfacher und ist wichtig für das Parsen von Programmiersprachen.

11.1. Pushdown-Automaten

Ziel: Einführung eines neuen Berechungsmodels: zustandsendlichen Automaten mit Zugriff auf einen Stack. Dieser kann eine unbegrenzte Menge an Information über die Historie der Berechnung speichern, diese aber nur auf sehr beschränktem Wege nutzen (*LIFO*: last in, first out).

Definition 11.1 (Syntax von PDAs)**.** Ein (nicht-deterministischer) Pushdown-Automat (N)PDA ist ein Tupel $M = (Q, \Sigma, \Gamma, q_0, \delta, Q_F)$ mit

- Q endliche Menge von Zuständen,
- $q_0 \in Q$ Initialzustand,
- $Q_F \subseteq Q$ Menge von Endzuständen,
- Σ (endliches) Eingabealphabet,
- Γ (endliches) Stack-Alphabet und
- $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \times \Gamma^* \times Q$ endliche Menge an Transitionen.

Wir schreiben Transitionen als $q_1 \xrightarrow[\gamma_1/\gamma_2]{\sigma} q_2$ statt $(q_1, \sigma, \gamma_1, \gamma_2, q_2) \in \delta$.

Um das Verhalten des PDA, seine Semantik, zu definieren benötigt wir den Begriff der Konfiguration, den Zustand des PDA zur Laufzeit. Dieser sollte enthalten:

- aktuellen Zustand,
- Inhalt des Stacks.

Definition 11.2 (Semantik eines PDAs)**.** Sei $M = (Q, \Sigma, \Gamma, q_0, \delta, Q_F)$. Die Menge der Konfigurationen von M ist $Q \times \Gamma^*$. Eine Konfiguration ist also von der Form (q, α) mit $q \in Q$ ein Kontrollzustand und $\alpha \in \Gamma^*$ der Stackinhalt. Der erste Buchstabe von α repräsentiert hierbei des untere Ende (Bottom-of-Stack), der letzte Buchstabe das obere Ende des Stacks (Top-of-Stack).

Die initiale Konfiguration ist (q_0, ε), der Automat ist also im Initialzustand und der Stack ist leer.

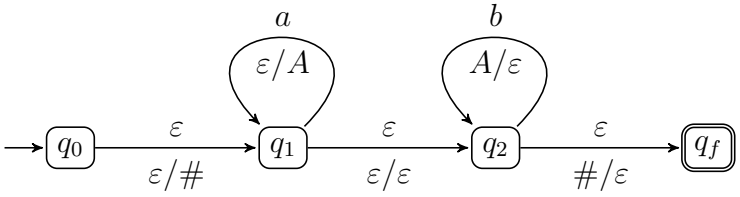
Wir nennen eine Konfigurationen final oder akzeptierend, wenn sie die Form $(q_f, \alpha) \in Q_F \times \Gamma^*$ hat, der Kontrollzustand also final und der Stackinhalt beliebig ist.

Die beschriftete Transitionsrelation zwischen Konfigurationen $\longrightarrow_M \subseteq (Q \times \Gamma^*) \times (\Sigma \cup \{\varepsilon\}) \times (Q \times \Gamma^*)$ ist definiert durch: $(q_1, \alpha_1) \xrightarrow{a}_M (q_2, \alpha_2)$, falls $q_1 \xrightarrow[\gamma_1/\gamma_2]{a} q_2$ und $\alpha_1 = \alpha.\gamma_1$, $\alpha_2 = \alpha.\gamma_2$ für $\alpha \in \Gamma^*$. Eine solche Transition kann also angewandt werden, wenn γ_1 das oberste Symbol des Stacks ist. In diesem Fall entfernt sie das Symbol γ_1 (es wird „gepopt“) und legt das Wort γ_2 auf den Stack (dieses wird „gepusht“).

Wir erweitern diese beschriftete Transitionsrelation auf Wörter aus Σ^*, wir schreiben also $(q, \alpha) \xrightarrow{w} (q', \alpha')$ wenn es eine Sequenz von Konfigurationen und mit $w_1, w_2, \ldots, w_{|w|}$ beschriftete Transitionen zwischen diesen gibt.

Die Sprache von M ist $L(M) := \{w \in \Sigma^* \mid (q_0, \varepsilon) \xrightarrow{w}_M^* (q_f, \alpha) \in Q_F \times \Gamma^*\}$.

Beispiel 11.3. M:



Es gilt $L(M) = \{a^n b^n \mid n \in \mathbb{N}\}$.

Bemerkung. Ein PDA M induziert einen gewöhnlichen Automaten ohne Speicher, nämlich $A_M := (Q \times \Gamma^*, \Sigma, (q_0, \varepsilon), \rightarrow_M, Q_F \times \Gamma^*)$ mit $L(M) = L(A_M)$. A_M hat jedoch unendlich viele Zustände.

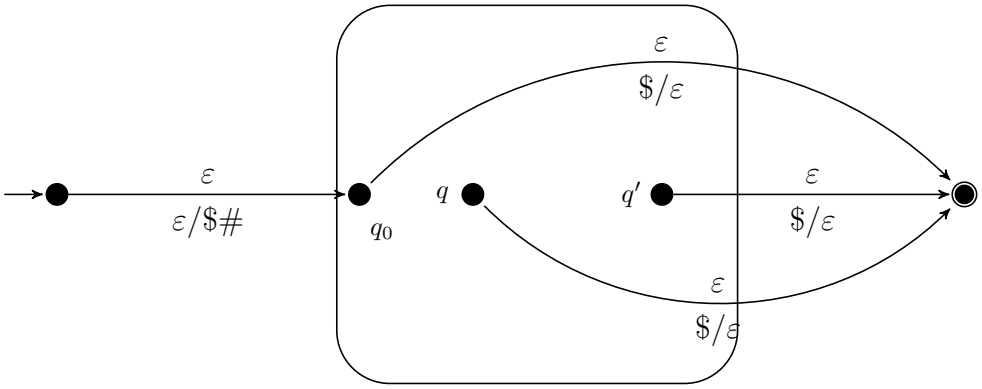
Akzeptanz mit leerem Stack. In der Literatur findet sich häufig eine alternative Definitionen von Pushdown-Automaten, nämlich Pushdown-Automaten, die mit *leerem Stack akzeptieren*. Ein solcher Automat ist gegeben durch $M = (Q, \Sigma, \Gamma, q_0, \#, \delta)$, wobei $\#$ ein *initiales Stack-Symbol* ist, und die anderen Komponenten wie zuvor. Der Automat startet in der Konfiguration $(q_0, \#)$, $\#$ befindet sich also initial auf dem Stack. Der Automat hat keine Finalzustände, sondern er akzeptiert in Konfigurationen der Form (q, ε), in denen der Stack leer ist.

Satz 11.4. Die Modelle für Pushdown-Automaten sind äquivalent: Zu einem Automaten gemäß eines Modells lässt sich ein Automat gemäß des anderen Modells konstruieren, der die selbe Sprache akzeptiert.

Beweisskizze:

- Angenommen ein Automat, der mit leerem Stack akzeptiert, ist gegeben. Wir imitieren dies mit unserer Definition. Sei # das initiale Stack-Symbol. Füge hinzu:
  - Neuen Initialzustand
  - Neuen Endzustand
  - Neues Stack-Symbol \$

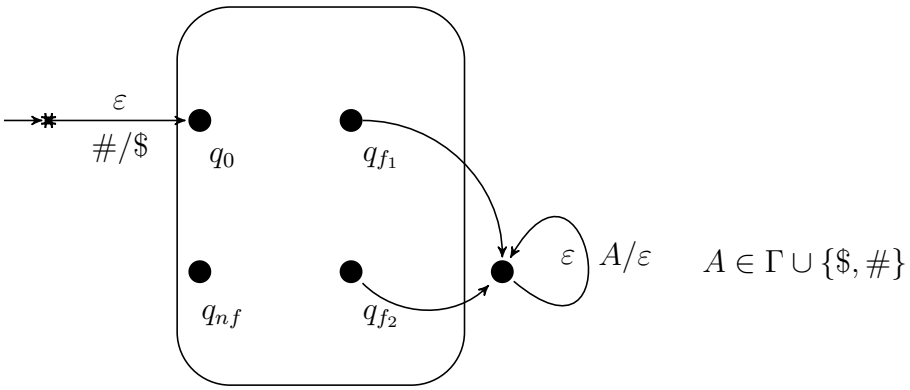
  Bild:



  Der Automat fügt am Anfang \$ als unterstes und # als zweit-unterstes Stacksymbol ein. Dann verhält er sich wie der gegebene Automat. Wenn der Stack des gegebenen Automaten leer ist, befindet sich nur noch \$ auf dem Stack. Zusätzliche Transitionen überprüfen dies, in dem sie \$ entfernen und in einen Endzustand wechseln

- Angenommen ein Automat, der durch Endzustände akzeptiert, ist gegeben. Wir imitieren dies durch Akzeptanz mit leerem Stack. Wir fügen ein neues Stack-Symbol \$ hinzu, um „versehentliche Akzeptanz“ bei leerem Stack zu verhindern. Von Endzuständen des ursprünglichen Automaten kann man in einen speziellen Zustand gehen, in dem der Stack geleert wird.

  Bild:



Die erste Transition ersetzt das Initiale Stacksymbol # durch \$, welches von den Transitionen des gegebenen Automaten nicht verändert wird. Von Endzuständen aus kann man in einen zusätzlichen Zustand wechseln, in dem der Stack geleert wird.

□

Bemerkung. Bei beiden Modellen (Akzeptanz mit Endzuständen, Akzeptanz mit leerem Stack) kann man annehmen, dass alle Transitionen

- immer ein Symbol wegnehmen
- maximal zwei Symbole pushen

Um die erste Einschränkung zu garantieren, muss man annehmen, dass auch Automaten, die mit Endzuständen akzeptieren, mit einem initialen Symbol auf dem Stack starten.

11.2. Äquivalenz mit CFL

Ziel: Zeige dass die Sprachen, die von PDAs akzeptiert werden, genau die kontextfreien Sprachen sind.

Satz 11.5. Falls L kontextfrei ist, dann gibt es einen PDA M mit $L = L(M)$.

Beweis. Sei $L = L(G)$ mit $G = (N, \Sigma, P, S)$ in Greibach-Normalform, also Produktionen $A \to aB_1...B_k$ und potentiell $S \to \varepsilon$. Wir konstruieren einen PDA M, der mit leerem Stack akzeptiert. Dieser simuliert die Linksableitungen der Grammatik. Interessanterweise wird hierfür nur ein Zustand benötigt.

Definiere: $M := (\{q\}, \Sigma, N, q, S, \delta)$ wobei jede Produktion $A \to aB_1...B_k$ zu einer eine Transition $q \xrightarrow[A/B_k...B_1]{a} q$ führt. Wir lesen Buchstaben a der Eingabe, entfernen A vom Top-of-Stack und schreiben $B_1...B_k$ (in der richtigen Reihenfolge) auf den Stack. Falls $S \to \varepsilon$, dann auch $q \xrightarrow[S/\varepsilon]{\varepsilon} q$.

Es gilt $L(M) = L(G)$. □

Bemerkung. Es ist auch möglich eine ähnliche Konstruktion anzugeben, die nicht auf die Greibach-Normalform aufbaut. Dies ist eine gute Übung. Der Vorteil obiger Konstruktion ist, das in jedem Schritt genau ein Buchstabe des Eingabeworts gelesen wird.

Satz 11.6. Jede Sprache $L(M)$ eines PDA M ist eine kontextfreie Sprache.

Der Beweis verwendet die sogenannte **Tripel-Konstruktion**. Wir konstruieren eine Grammatik mit Nichtterminalen der Form (q, A, q'), wobei q, q' Kontrollzustände des PDA sind und A ein Stacksymbol. Intuitiv sollen sich von einem solchen Nichtterminal genau die Wörter w ableiten lassen, die M entlang einer Transitionssequenz lesen kann, die von Kontrollzustand q zu Kontrollzustand q' geht und dabei das Stacksymbol A vom Top-of-Stack entfernt. Wir möchten also, dass $(q, A, q') \Rightarrow^* w$ genau dann gilt, wenn es eine Sequenz von Konfigurationen

$$(q, \alpha.A) \xrightarrow{v_0} (q'', \alpha.\beta_1) \xrightarrow{v_1} (q_2, \alpha.\beta_2) \dots \xrightarrow{v_{k-1}} (q_k, \alpha.\beta_k) \xrightarrow{v_k} (q', \alpha)$$

gibt. Hierbei sind die $q'', q_2, \dots, q_k$ geeignete Zwischenzustände, $v_0 \dots v_k = w$ und die $\beta_1, \dots, \beta_k$ nicht-leere Stackinhalte. Manche der v_i können gleich ε sein. Man beachte, dass der untere Teil des Stacks, also Stackinhalt α, während dieser Berechnung nicht angefasst wird.

Beweis. Es sei $M = (Q, \Sigma, \Gamma, q_0, \#, \delta)$, der gegebene PDA, der mit leerem Stack akzeptiert.

Wir nehmen $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \times \Gamma^* \times Q$ an, d.h. der PDA liest in jedem Schritt genau ein Stacksymbol.

Wir definieren $G := ((Q \times \Gamma \times Q) \cup \{S\}, \Sigma, P, S)$, wobei P wie folgt definiert ist:

$$\begin{aligned} P = \quad & \{S \to (q_0, \#, q) \mid q \in Q\} \\ \cup \quad & \left\{ \begin{array}{l} (q, A, q') \to a(q'', B_m, q_1)(q_1, B_{m-1}, q_2)...(q_{m-1}, B_1, q') \\ \qquad \Big| \; q \xrightarrow[A/B_1 \dots B_m]{a} q'' \text{ mit } B_1 \dots B_m \neq \varepsilon, q_1, ..., q_{m-1} \in Q \end{array} \right\} \\ \cup \quad & \left\{ (q, A, q'') \to a \;\Big|\; q \xrightarrow[A/\varepsilon]{a} q'' \right\} . \end{aligned}$$

Die Regeln der Form $S \to (q_0, \#, q)$ raten den Zustand q, in dem M ist, wenn der Stack leer wird (also das initiale Stacksymbol und alle aus ihm resultierenden Symbole ersetzt wurden). Die Regeln der Form $(q, A, q') \to a(q'', B_m, q_1)(q_1, B_{m-1}, q_2)...(q_{m-1}, B_1, q')$ wenden eine Transition $q \xrightarrow[A/B_1 \dots B_m]{a} q''$ an, die den Top-of-Stack A durch $B_1 \dots B_m$ (in der richtigen Reihenfolge) ersetzt, und rät dabei für jedes B_i die Zustände q_{m-i} und q_{m-i+1}, in denen M ist, bevor bzw. nachdem B_i vom Stack entfernt wurde. Die Regeln der Form $(q, A, q'') \to a$ wenden eine Transition $q \xrightarrow[A/\varepsilon]{a} q''$ an, die A vom Top-of-Stack entfernt und in den gewünschten Kontrollzustand wechselt. In beiden Fällen gilt $a \in \Sigma \cup \{\varepsilon\}$.

Es gilt $L(M) = L(G)$. □

Bemerkung. Durch Betrachten des Automaten lassen sich bei der händischen Konstruktion der Regeln viele unnütze Produktionen ausschließen. Betrachte beispielsweise eine Transition $q \xrightarrow[\#/\#\#]{a} q''$ des Automaten, welche Regeln der Form $(q, \#, q') \to a(q'', \#, q_1)(q_1, \#, q')$ induziert.

Es ist nur möglich, ein Terminalwort aus $(q, \#, q')$ abzuleiten, wenn in Zustand q' Stacksymbole gelöscht werden können, es also mindestens eine Transition der Form $q''' \xrightarrow[X/\varepsilon]{b} q'$ gibt (mit q''' beliebiger Zustand, X beliebiges Stacksymbol). Wenn dies nicht der Fall ist, kann das Nichtterminal $(q, \#, q')$, alle seine Regeln und alle Regeln auf deren rechten Seite es vorkommt entfernt werden.

Als Zwischenzustand q_1 kommen ebenfalls nur Zustände in Frage, in denen Stacksymbole gelöscht werden können (wie oben). Außerdem muss es möglich sein, im Kontrollfluss des Automaten (d.h. ohne Berücksichtigung der Effekte der Transitionen auf dem Stack) von Zustand q_1 zu Zustand q' zu kommen. Für Zustände q_1, die eine dieser Bedingungen verletzen, ist es nicht nötig, die Regel $(q, \#, q') \to a(q'', \#, q_1)(q_1, \#, q')$ einzuführen.

Zuletzt sollte man, wenn man die Konstruktion von Hand durchführt, nur die Nichtterminale und Regeln einführen, die tatsächlich beim vom Startsymbol ausgehenden Ableitungen auftreten können.

Korollar 11.7. Die kontextfreien Sprachen sind genau die Sprachen, die als $L(M)$ für PDAs M auftreten.

Korollar 11.8. Zu jedem PDA M gibt es einen PDA M', der nur einen Kontrollzustand hat und $L(M) = L(M')$ erfüllt.

12. Abschlusseigenschaften von CFLs

12.1. Positive Resultate

Ziel: Zeige Abgeschlossenheit unter Vereinigung $\cup$, Konkatenation $.$, der Kleeneschen Hülle *, Homomorphismen h, inversen Homomorphismen h^{-1}, Substitutionen und regulärem Schnitt.

Satz 12.1. Die Klasse CFL ist abgeschlossen unter $\cup, ., ^*$.

Der Satz lässt sich ähnlich wie für reguläre Sprachen durch eine Konstruktion auf Grammatiken zeigen und wurde in der Übung diskutiert.

Definition 12.2 (Substitution)**.** Eine *Substitution* ist eine Funktion $\sigma : \Sigma \to \mathbb{P}(\Delta^*)$, die jedem Buchstaben $a \in \Sigma$ eine Sprache $\sigma(a)$ über Δ zuordnet. Sie heißt

i) *regulär*, falls $\sigma(a)$ regulär für alle $a \in \Sigma$

ii) *kontextfrei*, falls $\sigma(a)$ kontextfrei für alle $a \in \Sigma$

Die Anwendung von σ auf ein Wort $w \in \Sigma^*$ ergibt die Sprache:

$$\sigma(\epsilon) = \{\epsilon\}$$
$$\sigma(a_1, ..., a_n) = \sigma(a_1) \dots \sigma(a_n)$$

Die Anwendung von σ auf eine Sprache $L \subseteq \mathbb{P}(\Sigma^*)$ liefert

$$\sigma(L) = \bigcup \{\, \sigma(w) \mid w \in L \,\} \subseteq \Delta^*$$

Satz 12.3. Die kontextfreien (regulären) Sprachen sind abgeschlossen unter kontextfreien (regulären) Substitutionen.

Beweis. Wir beweisen exemplarisch, dass CFL unter kontextfreien Substitutionen abgeschlossen sind.

Sei $G = (N, \Sigma, P, S)$. Für jedes $a \in \Sigma$ sei $G_a = (N_a, \Delta, P_a, S_a)$ eine Grammatik für $\sigma(a)$, wobei alle Mengen der Nicht-Terminale paarweise disjunkt sein mögen. Ersetze in P jedes Vorkommen von a in einer Produktion durch S_a, das liefert eine neue Menge P' von Produktionen. Die Grammatik $\sigma(G) = (\bar{N}, \Delta, \bar{P}, S)$ mit

$$\bar{N} = N \cup \bigcup_{a \in \Sigma} N_a \quad \text{und} \quad \bar{P} = P' \cup \bigcup_{a \in \Sigma} P_a$$

erfüllt dann $\sigma(L(G)) = L(\sigma(G))$:

$\subseteq$: Für $v \in \sigma(L(G))$ existiert ein $u = u_1 \dots u_n \in L(G)$ mit $v \in \sigma(u) = \sigma(u_1) \dots \sigma(u_n)$. Folglich existiert eine Zerlegung $v = v_1 \dots v_n$ mit $v_i \in \sigma(u_i)$ für $1 \leq i \leq n$. Betrachte eine G-Ableitung $S \Rightarrow^* u$. Ersetzt man darin alle Vorkommen von $a \in \Sigma$ durch S_a, so erhält man eine Ableitung $S \Rightarrow^* S_{u_1} \dots S_{u_n}$ in $\sigma(G)$ ohne Terminalsymbole. Diese wird kombiniert mit Ableitungen $S_{u_i} \Rightarrow^* v_i$, $1 \leq i \leq n$, was eine $\sigma(G)$-Ableitung $S \Rightarrow^* v$ liefert.

$\supseteq$: Betrachte eine $\sigma(G)$-Ableitung $S \Rightarrow^* v \in \Delta^*$, aufgeschrieben als Baum, mit S als Wurzel oben, den Buchstaben von v in den Blättern unten, und allen Knoten auf unterschiedlichen Höhen, was dem Zeitpunkt der Anwendung der passenden Produktion entspricht.

Nach Konstruktion von $\sigma(G)$ tritt in jedem Ast von einem Blatt zur Wurzel genau ein Nicht-Terminal der Form S_a mit $a \in \Sigma$ auf. Genauer, v erlaubt eine Zerlegung $v = v_1 \dots v_n$ in Teilwörter mit $S_{u_i} \Rightarrow^* v_i$ und somit $v_i \in \sigma(u_i)$ für geeignete Buchstaben $u_i \in \Sigma$.

Aufgrund der Kontextfreiheit lassen sich die Knoten in verschiedenen Zweigen des Ableitungsbaums unabhängig voneinander in der Höhe (=Zeit) verschieben. Damit läßt sich der Baum, bzw. die Ableitung, so umordnen, dass zunächst nur Produktionen angewendet werden, die von G herrühren. Das liefert eine G-Ableitung $S \Rightarrow^* u_1 \dots u_n$ und parallel nebeneinander Ableitungen $S_{u_i} \Rightarrow^* v_i$, woraus wir $v \in \sigma(u_1 \dots u)n) \subseteq \sigma(L(G))$ folgern. □

Korollar 12.4. Kontextfreie Sprachen sind abgeschlossen unter homomorphen Bildern.

Beweis. Wir wissen, dass ein Homomorphismus $h\colon \Sigma^* \to \Delta^*$ eindeutig durch seine Einschränkung auf die Buchstaben $h_{|\Sigma}$ definiert ist. Diese können wir als die kontextfreie Substitution auffassen, die jedem Buchstaben a die einelementige Sprache $\{h_{|\Sigma}(a)\}$ zuordnet. □

Satz 12.5. Kontextfreie Sprachen sind abgeschlossen unter inversen Homomorphismen.

Beweis. Nutze automatentheoretische Konstruktion. Sei $L = L(M)$ für den PDA $M = (Q, \Delta, \Gamma, q_0, \delta, Q_F)$. Sei $h : \Sigma^* \to \Delta^*$. Konstruiere PDA M' für $h^{-1}(L)$.
Idee: Speichere nach dem Lesen der Eingabe $a \in \Sigma$ zunächst $h(a) \in \Delta^*$ im Kontrollzustand und baue dieses Wort sukzessive ab gemäß M.

Genauer: Definiere $M' := (Q', \Sigma, \Gamma, (q_0, \epsilon), \delta', Q_F \times \{\epsilon\})$ mit Zustandsmenge

$$Q' := \{(q, x) \mid q \in Q, x \text{ Suffix von } h(a) \text{ für ein } a \in \Sigma\}$$

Die Übergänge sind gegeben durch

$$\begin{aligned}
\delta' : (q, \epsilon) &\xrightarrow[\epsilon,\epsilon]{a} (g, h(a)) && \text{für alle } a \in \Sigma \\
(q, bx) &\xrightarrow[v_1,v_2]{\epsilon} (p, x) && \text{falls } q \xrightarrow[v_1,v_2]{b} p \in \delta \\
(q, x) &\xrightarrow[v_1,v_2]{\epsilon} (p, x) && \text{falls } q \xrightarrow[v_1,v_2]{\epsilon} p \in \delta
\end{aligned}$$

wobei $b \in \Delta$, $x \in \Delta^*$, $v_i \in \Gamma^*$ mit $|v_1| \leq 1$.

Es gilt $L(M') = h^{-1}(L)$. Nach Konstruktion akzeptiert M' ein Wort $u = u_1 \dots u_n \in \Sigma^*$ genau dann, wenn M das Wort $h(u) = h(u_1) \dots h(u_n)$ akzeptiert. □

Satz 12.6. Falls $L \subseteq \Sigma^*$ kontextfrei und $R \subseteq \Sigma^*$ regulär, dann ist $L \cap R$ kontextfrei.

Ähnlich wie im Beweis von Satz 11.6 nutzt der Beweis die Techniken der Summarisierung.

Beweis. Wir gehen davon aus, dass $L = L(G)$ für eine kontextfreie Grammatik G in Chomsky-Normalform. Potentiell gibt es die Produktion $S \to \epsilon$, dann erscheint aber S nicht auf der rechten Seite einer Produktion.

Sei $R = L(A)$ mit $A = (Q, \Sigma, q_0, \to, Q_F)$.

Konstruiere die Grammatik $G' = (N', \Sigma, P', S')$ für $L \cap R$ wie folgt:

Nicht-Terminale: $N' := (Q \times N \times Q) \cup \{S'\}$, wobei S' ein neues Symbol ist.

Produktionen:

$$\begin{aligned}
S' &\to \epsilon && \text{falls } \epsilon \in L \cap R; \\
S' &\to (q_0, S, q_f) && \text{für alle } q_f \in Q_F; \\
(q_1, X, q_2) &\to a && \text{falls } q_1 \xrightarrow{a} q_2 \text{ und } X \to a; \\
(q_1, X, q_2) &\to (q_1, Y, q')(q', Z, q_2) && \text{falls } X \to YZ \text{ und } q' \in Q.
\end{aligned}$$

Intuitiv lassen sich aus dem Symbol (q_1, X, q_2) alle Wörter w mit $X \Rightarrow^* w$ in G und $q_1 \xrightarrow{w} q_2$ in A ableiten. Für Produktionsregeln der Form $X \to YZ$ setzt die Konstruktion alle möglichen Zwischenzustände, die erreicht werden können, ein. □

12.2. Negative Resultate

Satz 12.7. Die Klasse CFL ist nicht abgeschlossen unter Schnitt $\cap$ und Komplementierung $^-$.

Beweis. Betrachte die Sprachen $L_1 = \{a^n b^m c^m \mid n, m \in \mathbb{N}\}$, $L_2 = \{a^n b^n c^m \mid n, m \in \mathbb{N}\}$. Beide sind kontextfrei, ihr Schnitt $L_1 \cap L_2 = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ jedoch nicht.

Wäre CFL abgeschlossen unter Komplement $^-$, dann wäre CFL aufgrund des Abschlusses unter Vereinigungen $\cup$ und den De-Morganschen-Regeln auch abgeschlossen unter Schnitt $\cap$, denn $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$. Dies widerspricht dem ersten Teil des Beweises. □

13. Pumping-Lemma für CFGs

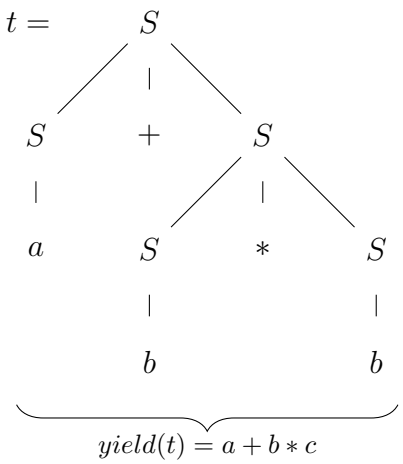
Ziel: Einführung eines Pumping-Lemmas, das uns erlaubt zu zeigen, dass eine Sprache nicht kontextfrei ist.

13.1. Parse-Bäume

Definition 13.1. Ein *Parse-Baum* von einem Nichtterminal A in $G = (N, \Sigma, P, S)$ ist ein Baum mit folgenden Eigenschaften:

- Jeder Knoten ist mit einem Symbol aus $N \cup \Sigma \cup \{\epsilon\}$ beschriftet. Die Wurzel mit A und jeder innere Knoten ist mit einem Symbol aus N beschriftet.
- Falls B ein innere Knoten mit k Kindern $B_1 \dots B_k$ (von links nach rechts), dann
  - $B_1, \dots, B_k \in N \cup \Sigma$ und $B \to B_1 \dots B_k$
  - oder $k = 1, B_1 = \epsilon$ und $B \to \epsilon \in P$.

Der *yield* eines Parse-Baumes t, mit $\text{yield}(t) \in (N \cup \Sigma)^*$ bezeichnet, ist die Konkatenation der Blätter von links nach rechts.

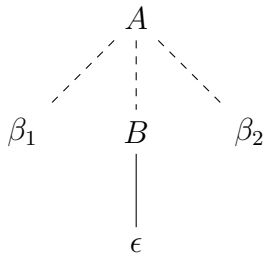


Definition 13.2. Ein *Parse-Baum assoziiert mit einer Ableitung* $A = \alpha_0 \Rightarrow \dots \Rightarrow \alpha_n$ ist definiert durch Induktion über die Länge der Ableitung:

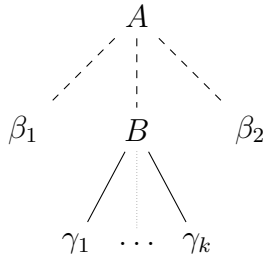
$n = 0$**:** Für die leere Ableitung von A nach A, erhalten wir den Parse-Baum mit dem einzelnen Knoten (Wurzel und Blatt) A.

Abbildung 13.1.: Erweiterung des Parse-Baumes

(a) Falls $\gamma = \epsilon$, erhalten wir



(b) Falls $\gamma = \gamma_1 \dots \gamma_k$ mit $\gamma_i \in N \cup \Sigma$, erhalten wir



$n \to n+1$**:** Betrachte

$$\underbrace{A}_{\alpha_0} \to \cdots \to \underbrace{\beta_1 B \beta_2}_{\alpha_n} \to \underbrace{\beta_1 \gamma \beta_2}_{\alpha_{n+1}}$$

Sei nun t der Parse-Baum für $A \to \cdots \to \beta_1 B \beta_2$. Wir erweitern t abhängig von $B \to \gamma$, wie in Abbildung 13.1 gezeigt.

Satz 13.3. Sei $G = (N, \Sigma, P, S), A \in N$ und $\alpha \in (N \cup \Sigma)^*$.

a) $A \Rightarrow^* \alpha$ genau dann wenn es einen Parse-Baum t von A aus gibt mit $yield(t) = \alpha$

b) Für jede Ableitung $A \Rightarrow^* \alpha$ gibt es eine eindeutigen Parse-Baum, nämlich den damit assoziierten.

c) Ein Parse-Baum kann mit mehreren Ableitungen assoziiert sein, aber nur mit einer Linksableitung und einer Rechtsableitung.

13.2. Das Pumping-Lemma

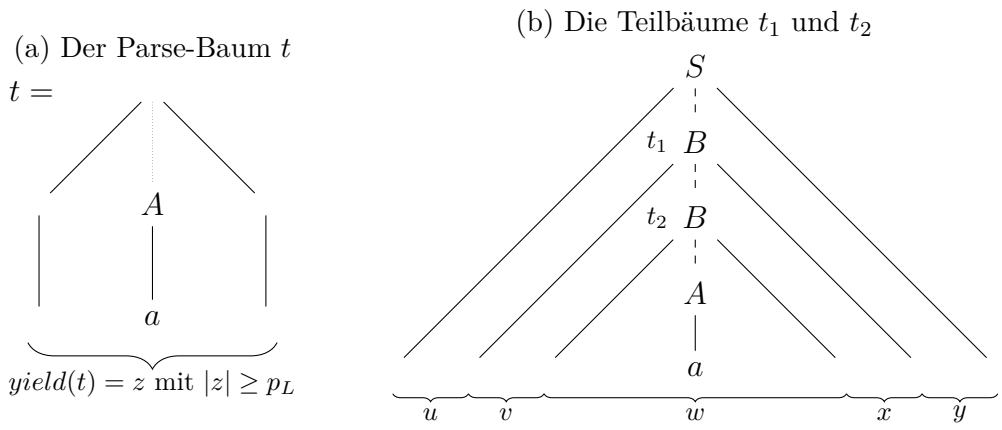
Ziel: Klassisches Pumping-Lemma für kontextfreie Sprachen.

Satz 13.4 (Bar-Hillel, Perles, Shamir '61)**.** Sei L eine CFL. Es gibt eine Konstante p_L sodass für alle $|z| \geq p_L$ eine Zerlegung $z = uvwxy$ existiert mit

1. $|vx| \geq 1$
2. $|vwx| \leq p_L$
3. $uv^iwx^iy \in L$ für alle $i \in \mathbb{N}$

Beweis. Sei G eine CFG in CNF für $L \setminus \{\epsilon\}$ und sei k die Zahl der Nichtterminale in G. Wir definiere: $p_L := 2^k$. Nun betrachten wir $z \in L$ mit $|z| \geq p_L$. Der Parse-Baum von z ist (wegen CNF) ein Binärbaum bis auf den letzten Schritt $A \to a$.

Da der Baum $\geq 2^k$ Blätter hat und da der letzte Schritt $A \to a$, hat er Höhe mindestens $k+1$. Wähle einen längsten Pfad (dieser hat mindestens $k+2$ Knoten).



Auf diesem Pfad gibt es mindestens $k+1$ Nichtterminale. Da es nur k Nichtterminale gibt, müssen sich auf Grund des Taubenschlagprinzips Nichtterminale wiederholen. Nun betrachten wir die Wiederholung, die am Dichtesten an den Blättern liegt, wie in Teil (b) der Abbildung.

Das obere B liegt höchstens $k+1$ Schritte von den Blättern entfernt und wir nennen den Teilbaum ab dieser Stelle t_1. Das heißt, Teilbaum t_1 hat *yield* der Länge $\leq 2^k$. Sei t_2 Teilbaum am unteren B mit $yield(t_2) = w$. Nun gilt:
$yield(t_1) = vwx$, für geeignete $v, x \in \Sigma^*$ mit $|vwx| \leq 2^k = p_L$. Es gilt $v \neq \epsilon$ oder $x \neq \epsilon$. Um das zu sehen, betrachte die erste Ableitung $B \to CD$, die auf das obere B angewandt wird. Nun liegt t_2 vollständig in dem Baum von C oder dem Baum von D. Im ersten Fall ist x von D abgeleitet, im zweiten Fall ist v von C abgeleitet. Also $x \neq \epsilon$ oder $v \neq \epsilon$. Zuletzt betrachte $B \Rightarrow^* vBx$ und $B \Rightarrow^* w$. Also $B \Rightarrow^* v^i w x^i$ für alle $i \in \mathbb{N}$. □

Beispiel 13.5. $L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ ist nicht kontextfrei.

Beweis. Angenommen L wäre kontextfrei. Sei p_L die Pumping-Konstante. Betrachte $a^{p_L} b^{p_L} c^{p_L} \in L$. Wir zerlegen das Wort in $uvwxy$, sodass die Bedingungen des Pumping-Lemmas gelten. Da $|vwx| \leq p_L$, kann es nicht sein, dass v und x sowohl a's als auch c's enthält.

Falls v und x nur aus a's bestehen, betrachte $uv^0wx^0y = uwy$. Dieses Wort besteht aus p_L vielen b's und c's, aber nur aus $< p_L$ vielen a's, da $|vx| \geq 1$ ↯.

Falls vx nur aus bs, nur aus cs, nur aus as und bs, oder nur aus bs und cs besteht folgt der Widerspruch analog. □

Beispiel 13.6. Die Copy-Sprache $L = \{ww \mid w \in \{a, b\}^*\}$ ist nicht kontextfrei.

Beweis. Angenommen L wäre kontextfrei. Sei p_L die Pumping-Konstante. Betrachte

$$z = \underbrace{a^{p_L}}_{z_1} \underbrace{b^{p_L}}_{z_2} \underbrace{a^{p_L}}_{z_3} \underbrace{b^{p_L}}_{z_4} \in L \ .$$

Wir zerlegen das Wort in $uvwxy$, sodass die Bedingungen des Pumping-Lemmas gelten. Da $|vwx| \leq p_L$, kann es nicht sein, dass v und x sowohl Teile von z_1 als

auch Teile von z_3 bzw. Teile von z_2 und Teile von z_4 enthält. Analog zum obigen Beispiel kann man zeigen, dass geeignetes Pumpen zu einem Wort führt, dass nicht in L enthalten ist. □

Bemerkung. Es gibt Sprachen, die das Pumping-Lemma erfüllen, obwohl sie nicht kontextfrei sind, z.B. die Sprache $\{a^n b^m c^k d^\ell \mid m = 0 \text{ oder } m = k = \ell\}$. Um auch bei solchen Sprachen nachweisen zu können, dass sie nicht kontextfrei sind, kann man stärkere Versionen des Pumping-Lemmas, z.B. Ogdens Lemma betrachten.

14. Entscheidungsverfahren für CFLs

Ziel: Untersuche algorithmische Probleme für CFLs.

14.1. Positive Resultate

Das Wortproblem

Wortproblem für kontextfreie Sprachen (WORDCFL)	
Gegeben:	Kontextfreie Grammatik $G = (N, \Sigma, P, S)$, Terminalwort $w \in \Sigma^*$.
Frage:	Gilt $w \in L(G)$?

In Kapitel 9 haben wir bereits gesehen, dass das Wortproblem WORDCFL gelöst werden kann.

Leerheit

Leerheitsproblem für kontextfreie Sprachen (EMPTYCFL)	
Gegeben:	Kontextfreie Grammatik $G = (N, \Sigma, P, S)$.
Frage:	Gilt $L(G) = \emptyset$?

Satz 14.1. EMPTYCFL ist entscheidbar in $O(|P|^2)$.

Beweis. Berechne aufsteigende Kette von Mengen von Nicht-Terminalen $N_0 \subseteq N_1 \subseteq ...$ bis ein Fixpunkt $N_n = N_{n+1} = \bigcup_{i \in \mathbb{N}} N_i$ erreicht ist. N_i enthält die Nichtterminale, von denen aus sich ein Terminalwort durch einen Parsebaum der Höhe $i + 1$ ableiten lässt.

Formal definieren wir

$$N_0 = \{A \in N \mid A \to w \in P \text{ mit } w \in \Sigma^*\} ,$$
$$N_{i+1} = \{A \in N \mid A \to \alpha \in P \text{ mit } \alpha \in (N_i \cup \Sigma)^*\} .$$

Wir erhalten, dass die Sprache von G leer ist, genau dann wenn das Startsymbol S nicht Element von $\bigcup_{i \in \mathbb{N}} N_i$ ist.

Die Zeitkomplexität von $O(|P|^2)$ ergibt sich daraus, dass wir jede Produktion höchstens einmal nutzen müssen. □

Reguläre Inklusion

Reguläre Inklusion kontextfreier Sprachen (REGINCLUSIONCFL)	
Gegeben:	Kontextfreie Grammatik $G = (N, \Sigma, P, S)$, NFA A.
Frage:	Gilt $L(G) \subseteq L(A)$?

Satz 14.2. REGINCLUSIONCFL ist entscheidbar in $O(|G|^6 \cdot 2^{6|A|})$.

Beweis. Wir nutzen, dass $L(G) \subseteq L(A)$ gdw. $L(G) \cap \overline{L(A)} = \emptyset$. Wir determinisieren A und invertieren die Endzustände. Dies liefert einen DFA B mit $L(B) = \overline{L(A)}$. Die Potenzmengenkonstruktion benötigt hierbei exponentiell viel Zeit, .

CFLs sind effektiv abgeschlossen unter regulärem Schnitt. Wir können also eine kontextfreie Grammatik H konstruieren mit $L(H) = L(G) \cap L(B)$. Diese Grammatik kann nun als Instanz des Problems EMPTYCFL gesehen werden, wir überprüfen also $L(H) = \emptyset$. Falls $L(H) = \emptyset$ gilt $L(G) \subseteq L(A)$, ansonsten gibt es ein Wort aus $L(G) \setminus L(A)$ und die Inklusion ist verletzt.

Die Konstruktion von H gemäß Satz 12.6 führt folgendes ein:

- $|N|(2^{|Q|})^2$ Nichtterminale,
- $|\Sigma| \cdot |N| \cdot (2^{|Q|})^2$ Produktionen der Form $(q_1, X, q_2) \to a$ (für $X \to a$),
- $|N|^3 \cdot (2^{|Q|})^2$ Produktionen der Form $(q_1, X, q_2) \to (q_1, Y, q')(q', Z, q_2)$ (für $X \to YZ$).

Insgesamt benötigt die Konstruktion eine Zeit von $O(|G|^3 \cdot 2^{3|A|})$. Das Anwenden des Tests auf Leerheit liefert $O((|G|^3 \cdot 2^{3|A|})^2) = O(|G|^6 \cdot 2^{6|A|})$. □

Endlichkeit

Endlichkeit kontextfreier Sprachen (FINITECFL)	
Gegeben:	Kontextfreie Grammatik $G = (N, \Sigma, P, S)$.
Frage:	Besteht $L(G)$ aus nur endlichen vielen Wörtern?

Solche Beschränktheitsprobleme sind von Bedeutung für die Verifikation. Um ein C-Programm in Hardware zu verwandeln, müssen wir prüfen, ob der Stack in der Höhe beschränkt ist und ob nur eine endliche Menge Speicher allokiert wird.

Intuitiv müssen wir überprüfen, ob es eine Schleife (Pumping) in G gibt, die verwendet werden kann, um immer längere Wörter zu erzeugen.

Beweis. Ein nicht optimaler Algorithmus kann mit Hilfe des Pumping-Lemmas konstruiert werden. Wir konvertieren die Grammatik G zu einer Grammatik in CNF $G' = (N', \Sigma, P', S')$. Sei $k = |N'|$ in G' und definiere $p_L := 2^k$. Prüfe, ob $L(G)$ ein Wort w enthält mit Länge $p_L \leq |w| \leq 2p_L$. Falls ja, return *false*, die Sprache ist unendlich, denn w erfüllt die Bedingungen des Pumping-Lemma. Falls nein, return *true*, die Sprache ist endlich.

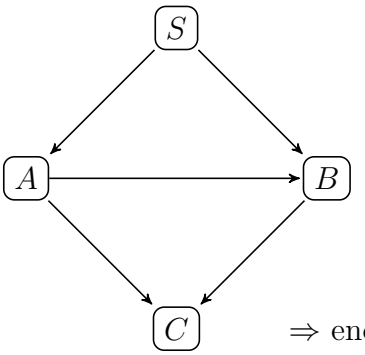
Um alle Wörter w mit $p_L \leq |w| \leq 2p_L$ zu überprüfen, verwenden wir wiederholt den CYK-Algorithmus.

Wir argumentieren, dass Algorithmus korrekt ist: Sei u das kürzeste Wort in $L(G)$ mit $|u| \geq p_L$. Behauptung: $|u| \leq 2p_L$. Angenommen $|u| > 2p_L$. Mit Pumping-Lemma $u = x_1x_2x_3x_4x_5$ wobei $|x_2x_3x_4| \leq p_L$ und $x_1x_3x_5 \in L(G)$. Da $|x_1x_2x_3x_4x_5| > 2p_L$ und $|x_2x_3x_4| \leq p_L$ folgt $|x_1x_3x_5| \geq p_L$ □

Um einen besseren Algorithmus zu erhalten überführen wir G in CNF G' für $L(G) \setminus \{\epsilon\}$ ohne unnütze Nicht-Terminale. Es gilt $L(G)$ endlich *gdw.* $L(G')$ endlich. Sei $G' = (N, \Sigma, P, S)$. Konstruiere gerichteten Graphen (V, E) mit $V = N$, also den Nichtterminalen als Knoten, und $E := \{A \to B \mid A \to BC \in P \text{ oder } A \to CB \in P\}$. Behauptung: $L(G')$ ist endlich *gdw.* (V, E) azyklisch ist. Gilt, da die Nichtterminale einer Schleife garantiert ein Wort ableiten (nicht unnütz) und das Wort nicht ϵ ist (da CNF).

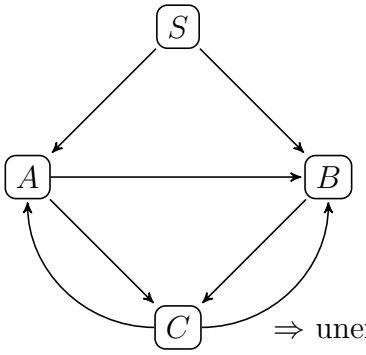
Beispiel 14.3. Sei $G = (\{S, A, B, C\}, \{a, b\}, P, S)$ mit P:

$$\begin{aligned} S &\to AB \\ A &\to BC \mid a \\ B &\to CC \mid b \\ C &\to a \end{aligned}$$



⇒ endlich.

$$\begin{aligned} S &\to AB \\ A &\to BC \mid a \\ B &\to CC \mid b \\ C &\to AB \end{aligned}$$



⇒ unendlich.

14.2. Negative Resultate

Universalität und Regularität. Wir haben gesehen, dass die Inklusion $L(G) \subseteq L(A)$ einer kontextfreien Sprache in einer regulären Sprache algorithmisch entscheidbar ist. Die „umgedrehte“ Inklusion $L(A) \subseteq L(G)$ ist nicht entscheidbar. Tatsächlich gibt es bereits eine feste reguläre Sprache, nämlich Σ^*, so dass $\Sigma^* \subseteq L(G)$ (welche die Gleichheit $\Sigma^* = L(G)$ implizieren würde) nicht entscheidbar ist.

Universalitätsproblem für kontextfreie Sprachen (UNIVERSALITYCFL)	
Gegeben:	Kontextfreie Grammatik $G = (N, \Sigma, P, S)$.
Frage:	Gilt $\Sigma^* = L(G)$?

Daraus folgt auch, das viele andere Probleme unentscheidbar sind, darunter nicht nur die Inklusion $L(G_1) \subseteq L(G_2)$ zwischen kontextfreien Sprachen, sondern auch Regularität.

Regularität kontextfreier Sprachen (REGULARITYCFL)	
Gegeben:	Kontextfreie Grammatik $G = (N, \Sigma, P, S)$.
Frage:	Ist $L(G)$ regulär?

Wir können an dieser Stelle keinen Beweis geben, argumentieren aber informell wie folgt, dass die Unentscheidbarkeit von UNIVERSALITYCFL die Unentscheidbarkeit von REGULARITYCFL impliziert. Angenommen wir hätten ein Verfahren, dass als Eingabe eine kontextfreie Grammatik G erwartet, und als Ausgabe entweder „$L(G)$ nicht regulär“ oder einen NFA A mit $L(G) = L(A)$ liefert. Um nun das Universalitätsproblem für eine Grammatik G zu entscheiden könnten wir dieses Verfahren verwenden. Entweder liefert es, dass $L(G)$ nicht regulär ist, wodurch auch $L(G) \neq \Sigma^*$ gilt, da Σ^* regulär ist, oder es liefert einen NFA A mit $L(G) = L(A)$. Da das Universalitätsproblem für reguläre Sprachen entscheidbar ist, könnten wir nun überprüfen, ob $L(A) = \Sigma^*$ und damit $L(G) = \Sigma^*$ gilt.

Schnittleerheit. Wir haben bereits gezeigt, dass der Schnitt von zwei kontextfreien Sprachen nicht notwendigerweise kontextfrei ist. Tatsächlich ist es sogar unmöglich, zu überprüfen ob der Schnitt zweier kontextfreier Sprachen ein Wort enthält.

Schnittleerheit kontextfreier Sprachen (INTERSECTIONEMPTINESSCFL)	
Gegeben:	Kontextfreie Grammatiken G_1, G_2.
Frage:	Gilt $L(G_1) \cap L(G_2) = \emptyset$?

Auch hier geben wir statt eines formalen Beweises eine intuitive Begründung. Pushdown-Automaten haben unendlich viele Konfigurationen, da sie auf ihrem Stack unbeschränkt viele Information speichern können. Dass sich trotzdem viele Probleme lösen lassen, liegt daran, dass dieser unbegrenzte Speicher nur auf eine sehr beschränkte Art und Weise verwenden lässt: Man kann nur auf das oberste Element zugreifen. Wenn man auf die unteren Elemente des Stacks zugreifen möchte, muss man zunächst die oberen Elemente entfernen, wodurch man sie „vergisst“.

Der Schnitt zweiter kontextfreien Sprachen entspricht zwei synchron laufenden Pushdown-Automaten, oder alternativ einem Pushdown-Automaten, der zwei separate Stacks als Speicher hat. Dies ist ein sehr viel mächtigeres Berechnungsmodell: Ein solcher Automat kann auf die unteren Stackeinträge zugreifen, ohne dabei die oberen zu vergessen, in dem die oberen Einträge vom einen Stack entfernt, dabei aber gleichzeitig zum anderen Stack hinzugefügt werden.

Satz 14.4. Die Probleme UNIVERSALITYCFL, REGULARITYCFL und INTERSECTIONEMPTINESSCFL sind unentscheidbar.

Es gibt also kein Computerprogramm, dass diese Probleme löst. Um den Satz zu beweisen, muss man zunächst ein formales Modell von Computerprogrammen erstellen

und zeigen, dass nicht alle wohlspezifizierten Berechnungsprobleme von Computerprogrammen gelöst werden können. Dies ist Gegenstand der Vorlesung „Theoretische Informatik 2“, in der Sie auch den Beweis des obigen Satzes sehen werden.